Treball Final de Grau

GRAU DE MATEMÀTIQUES

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

Fundamentos de Programación Lógica

Autor: Héctor Barriga Martín

Director: Dr. Juan Carlos Martínez Alonso
Departament de Matemàtiques

Barcelona, 27 de junio de 2018

Abstract

Logic Programming arises from the fundamental idea that First Order Logic can be used as a programming language.

The aim of the first two sections of this work is to present the theoretical foundations of Logic Programming and particularly of the SLD–Resolution Method. In the last section we show the Prolog programming language and give some examples where it shines over imperative programming languages.

Resumen

La Programación Lógica nace de la idea de que la lógica de primer orden puede ser utilizada como lenguaje de programación.

El objetivo de las dos primeras secciones de este trabajo es presentar los fundamentos teóricos de la Programación Lógica y muy particularmente de la SLD–Resolución. En la última sección mostramos el lenguaje de programación Prolog y damos algunos ejemplos en los que destaca por encima de los lenguajes de programación imperativos.

Agradecimientos

A mi tutor, por su seguimiento constante y sus valiosas correcciones. A todos mis profesores, por la mochila de conocimiento que me han ayudado a construir. A Yue, por su sonrisa, su sinceridad, su buen corazón, su paciencia infinita y su apoyo incondicional durante todo este tiempo. A mi madre, por soportarme todos estos años y por respetar mis decisiones.

Índice

Introducción

La Programación Lógica surge en la década de los 70 como resultado de los avances en Inteligencia Artificial y Demostración Automática que se produjeron en las décadas anteriores.

Desde las aportaciones de Herbrand en la década de los 30, la Demostración Automática se desarrolló hasta culminar en 1965 con la regla de Resolución de Robinson. La Resolución es una regla de inferencia para la lógica de primer orden especialmente apropiada para ser implementada en un ordenador.

En 1972 Kowalski y Colmerauer, considerados los padres de la Programación Lógica, llegan a la conclusión de que la lógica de primer orden puede ser utilizada como lenguaje de programación. Crean el lenguaje de programación Prolog (PROgramación LÓGica) e implementan el primer intérprete de Prolog en el lenguaje de programación ALGOL–W.

Más tarde van Emden y Kowalski desarrollan las semánticas declarativa, de punto fijo y procedural de los programas lógicos y demuestran que todas ellas son equivalentes.

En esta memoria presentamos los fundamentos teóricos de la Programación Lógica.

En la primera sección introducimos todos los conceptos necesarios de la lógica de primer orden, de los programas lógicos y de los modelos de Herbrand, mostramos las semánticas declarativa y de punto fijo y demostramos su equivalencia.

La segunda sección está dedicada en exclusiva a la SLD–Resolución, el refinamiento de la Resolución de Robinson que implementan los intérpretes de Prolog. Demostramos su corrección y completud.

En la tercera y última sección mostramos el lenguaje de programación Prolog y damos algunos ejemplos en los que destaca por encima de los lenguajes de programación imperativos.

1. Semántica de los Programas Lógicos

La Programación Lógica se basa en la lógica de primer orden. En esta primera sección introduciremos todos los conceptos de la lógica de primer orden que utilizaremos a lo largo de la memoria, definiremos lo que es un programa lógico y daremos dos semánticas equivalentes.

1.1. Lógica de primer orden

Definición 1.1 (Alfabeto de un lenguaje de primer orden)**.** El *alfabeto de un lenguaje de primer orden* $\mathcal{A}_S$ contiene los siguientes símbolos:

1. $x, y, z, x_0, x_1, \ldots$ (variables);
2. $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$ (conectivas de negación, conjunción, disyunción, implicación y doble implicación resp.);
3. $\exists, \forall$ (cuantificadores);
4. $(,)$ (paréntesis);
5. *a*) para cada $n \geq 1$ un conjunto (quizá vacío) de símbolos de función n–aria;
   b) para cada $n \geq 1$ un conjunto (quizá vacío) de símbolos de relación n–aria;
   c) un conjunto (quizá vacío) de símbolos de constante.

Denotaremos por $\mathcal{A}$ el conjunto de símbolos de (1) hasta (4) y por S el conjunto de símbolos de (5), de modo que $\mathcal{A}_S = \mathcal{A} \cup S$. El conjunto S puede ser vacío. Los símbolos de S deben no repetirse y ser distintos a los de $\mathcal{A}$. Nosotros utilizaremos la siguiente notación:

- $c, d, \ldots, c_0, c_1, \ldots$ para los símbolos de constante;
- $R, S, \ldots, R_0, R_1, \ldots$ para los símbolos de relación;
- $f, g, \ldots, f_0, f_1, \ldots$ para los símbolos de función.

Fijado un alfabeto $\mathcal{A}_S$, podemos definir los conjuntos de términos y fórmulas asociados.

Definición 1.2 (Término)**.** Definimos recursivamente un *término* como sigue:

1. Las variables de $\mathcal{A}$ son términos.
2. Los símbolos de constante de S son términos.
3. Si f es un símbolo de función n–ario de S y $t_1, \ldots, t_n$ son términos entonces $ft_1 \ldots t_n$ es término.

Denotaremos por T_S al conjunto de términos formados por estas reglas. Llamaremos *términos básicos* a los términos construidos sin usar la regla 1, i.e. a los términos que no contengan variables.

Definición 1.3 (Fórmula)**.** Definimos recursivamente una *fórmula* como sigue:

1. Si R es un símbolo de relación n–ario de S y $t_1, \ldots, t_n$ son términos de T_S, entonces $Rt_1 \ldots t_n$ es fórmula.
2. Si φ es fórmula, entonces $\neg\varphi$ es fórmula.
3. Si φ, ψ son fórmulas y $* \in \{\wedge, \vee, \to, \leftrightarrow\}$, entonces $\varphi * \psi$ es fórmula.
4. Si φ es fórmula y x es una variable de $\mathcal{A}$, entonces $\forall x\varphi$ y $\exists x\varphi$ son fórmulas.

Denotaremos por L_S al conjunto de fórmulas formadas por estas reglas, cuyos elementos denotaremos por $\varphi, \psi, \chi, \xi, \ldots, \varphi_0, \varphi_1, \ldots$. A las fórmulas formadas por la regla 1, i.e. a aquellas fórmulas que no contengan conectivas ni cuantificadores de $\mathcal{A}$, las llamaremos *fórmulas atómicas*. Denotaremos por $At(L_S)$ al conjunto de fórmulas atómicas de L_S.

Definición 1.4 (Lenguaje de primer orden)**.** Definimos el *lenguaje de primer orden asociado al alfabeto* $\mathcal{A}_S$ como L_S.

Demos ahora unas definiciones sobre las variables que aparecen en una fórmula.

Definición 1.5. Una *ocurrencia* de una variable x en una fórmula φ es una aparición concreta de la variable en la fórmula.

- Definimos $\mathrm{Var}(\varphi) := \{x\colon x \text{ es una variable que ocurre en la fórmula } \varphi\}$.
- Si $Qx\psi$ es una ocurrencia del cuantificador Q en una fórmula, diremos que ψ es el *alcance* de esa ocurrencia del cuantificador Q.
- Una ocurrencia de una variable x es una *ocurrencia ligada* si x está dentro del alcance de algún cuantificador, en caso contrario la ocurrencia de x es *libre*.
- Las *variables libres* de una fórmula φ son las que tienen al menos una ocurrencia libre en φ.
- Si $x_1, \ldots, x_n$ son todas las variables que aparecen en un término t, lo escribiremos de forma compacta como $t(x_1, \ldots, x_n)$. Además denotaremos por $t[s_1, \ldots, s_n]$ al término que se obtiene al sustituir en t todas las ocurrencias libres de x_i por el término s_i para todo $i = 1, \ldots, n$.
- Si $x_1, \ldots, x_n$ son todas las variables libres que aparecen en una fórmula φ, lo escribiremos de forma compacta como $\varphi(x_1, \ldots, x_n)$. Además denotaremos por $\varphi[s_1, \ldots, s_n]$ a la fórmula que se obtiene al sustituir en φ todas las ocurrencias libres de x_i por el término s_i para todo $i = 1, \ldots, n$.
- Una *fórmula cerrada* o *sentencia* es una fórmula sin variables libres. Denotaremos por $Sen(L_S)$ al subconjunto de fórmulas cerradas de L_S.
- Una *fórmula básica* es una fórmula sin variables.

1.2. Programas lógicos

Definición 1.6 (Clausuras universal y existencial de una fórmula)**.** Si φ es una fórmula y $x_1, \ldots, x_n$ son las variables libres de φ, definimos la clausura universal $\forall(\varphi)$ de φ como la fórmula

$$\forall(\varphi) := \forall x_1 \ldots \forall x_n \varphi.$$

De manera similar, definimos la clausura existencial $\exists(\varphi)$ de la fórmula φ como la fórmula

$$\exists(\varphi) := \exists x_1 \ldots \exists x_n \varphi.$$

Definición 1.7 (Cláusula)**.**

- Un *literal* es una fórmula atómica φ (literal positivo) o su negación $\neg\varphi$ (literal negativo).
- Una *cláusula* es la clausura universal de una disyunción de literales, i.e. una fórmula de la forma
$$\forall(\chi_1 \vee \cdots \vee \chi_n)$$
donde $\chi_1, \ldots, \chi_n$ son literales.
- Definimos el conjunto de *miembros* de una cláusula $\varphi = \forall(\chi)$ como el conjunto
$$Mb(\varphi) = \{\psi \colon \psi \text{ es un literal de } \chi\}.$$

Notación 1.8. Sean $\varphi_1, \ldots, \varphi_k$ literales positivos y $\neg\psi_1, \ldots, \neg\psi_r$ literales negativos. Denotaremos por

$$\varphi_1, \ldots, \varphi_k \leftarrow \psi_1, \ldots, \psi_r$$

una cláusula que sea clausura universal de una disyunción de literales $\varphi_1, \ldots, \varphi_k$, $\neg\psi_1, \ldots, \neg\psi_r$.

Observación 1.9. La cláusula $\forall(\varphi_1 \vee \varphi_2 \vee \neg\psi_1 \vee \neg\psi_2)$, con $\varphi_1, \varphi_2, \psi_1, \psi_2$ literales positivos, la podemos denotar de 4 maneras diferentes:

$$\begin{aligned}
&\varphi_1, \varphi_2 \leftarrow \psi_1, \psi_2\\
&\varphi_1, \varphi_2 \leftarrow \psi_2, \psi_1\\
&\varphi_2, \varphi_1 \leftarrow \psi_1, \psi_2\\
&\varphi_2, \varphi_1 \leftarrow \psi_2, \psi_1.
\end{aligned}$$

De igual modo, la cláusula $\varphi_1, \varphi_2 \leftarrow \psi_1, \psi_2$ denota a $4! = 24$ cláusulas distintas:

$$\begin{aligned}
&\forall(\varphi_1 \vee \varphi_2 \vee \neg\psi_1 \vee \neg\psi_2)\\
&\forall(\varphi_1 \vee \varphi_2 \vee \neg\psi_2 \vee \neg\psi_1)\\
&\qquad\vdots\\
&\forall(\neg\psi_1 \vee \varphi_2 \vee \neg\psi_2 \vee \varphi_1)\\
&\qquad\vdots
\end{aligned}$$

Esto no será un problema puesto que, en general, todas las cláusulas que denotadas por

$$\varphi_1, \ldots, \varphi_k \leftarrow \psi_1, \ldots, \psi_r$$

son lógicamente equivalentes.

Observación 1.10. La notación introducida utiliza la flecha $\leftarrow$ similar a la de la implicación $\rightarrow$. Este hecho no es casual, se debe a que

$$\forall(\chi)$$

donde χ es una cláusula con literales $\varphi_1, \ldots, \varphi_k, \neg\psi_1, \ldots, \neg\psi_r$ es lógicamente equivalente a

$$\forall\left(\psi_1 \wedge \cdots \wedge \psi_r \rightarrow \varphi_1 \vee \cdots \vee \varphi_k\right).$$

Con todo lo que hemos visto hasta ahora podemos dar la definición de programa lógico y algunos conceptos asociados.

Definición 1.11 (Cláusula de programa)**.** Una *cláusula de programa* es una fórmula de la forma

$$\varphi \leftarrow \psi_1, \ldots, \psi_n$$

i.e. una cláusula con exactamente un literal positivo.
Llamamos a φ, el único literal positivo, la *cabeza* de la cláusula, y a $\psi_1, \ldots, \psi_n$ el *cuerpo* de la cláusula.

Definición 1.12 (Realización básica)**.** Dada una cláusula de programa

$$\varphi = \forall x_1 \ldots \forall x_n\big(\chi(x_1, \ldots, x_n)\big),$$

una cláusula de programa ψ será una realización básica de φ si es de la forma

$$\psi = \chi[t_1, \ldots, t_n]$$

donde $t_1, \ldots, t_n$ son términos básicos.

Definición 1.13 (Programa lógico)**.** Un *programa lógico* es un conjunto finito de cláusulas de programa.

Definición 1.14 (Objetivo)**.** Un *objetivo* es una fórmula de la forma

$$\exists(\psi_1 \wedge \cdots \wedge \psi_n)$$

donde las ψ_i son fórmulas atómicas.

Observación 1.15. Si $\psi = \exists(\psi_1 \wedge \cdots \wedge \psi_n)$ es un objetivo entonces su negación $\neg\psi$ es lógicamente equivalente a la fórmula

$$\forall(\neg\psi_1 \vee \cdots \vee \neg\psi_n),$$

que denotamos por

$$\leftarrow \psi_1, \ldots, \psi_n.$$

Las últimas definiciones de esta subsección:

Definición 1.16.

- Un *hecho* es una cláusula de programa de la forma

  $$\varphi = \varphi \leftarrow$$

  i.e. con cuerpo vacío. Semánticamente, los hechos afirman que su cabeza es un axioma.

- Denotaremos por $\square$ la *cláusula vacía* (cláusula sin cabeza ni cuerpo). Semánticamente, representa una contradicción. No es una cláusula de programa.

1.3. Interpretaciones y modelos

Definición 1.17 (Asignación)**.** Dado un conjunto no vacío U, una *asignación* α sobre U es una aplicación cuyo dominio son las variables de $\mathcal{A}$ y cuyo codominio es U.
Definimos α_a^x, con x variable de $\mathcal{A}$ y $a \in U$, como la asignación igual a α salvo en la imagen de la variable x, que pasa a ser $\alpha_a^x(x) = a$.

Definición 1.18 (Interpretación)**.** Una *interpretación* es una 5–tupla

$$I = (L_S, U, \alpha, \pi, \rho),$$

donde L_S es un lenguaje de primer orden, U es un conjunto no vacío que llamaremos universo, α una asignación, π es una aplicación que

- a cada símbolo de constante $c \in S$ le hace corresponder un elemento c^I de U, y
- a cada símbolo de función n–ario $f \in S$ le hace corresponder una aplicación $f^I \colon U^n \to U$,

y ρ es una aplicación que

- a cada símbolo de relación n–ario $R \in S$ le hace corresponder una relación n–aria $R^I \subseteq U^n$.

Si x es una variable y $a \in U$, definimos $I_a^x = (L_S, U, \alpha_a^x, \pi, \rho)$, es decir I_a^x es la interpretación I con la asignación α_a^x en vez de α.

Definición 1.19 (Evaluación de un término)**.** Definimos recursivamente la *evaluación* $t^I \in U$ de un término $t \in T_S$ bajo una interpretación $I = (L_S, U, \alpha, \pi, \rho)$ como sigue:

1. Si t es una variable de $\mathcal{A}$, entonces $t^I = \alpha(t)$.
2. Si t es un símbolo de constante de S, entonces $t^I = t^I$.
3. Si $t = ft_1 \dots t_n$, con f símbolo de función n–aria de S y $t_i \in T_S$, entonces $t^I = f^I t_1^I \dots t_n^I$.

Definición 1.20 (Fórmulas ciertas)**.** Sea $I = (L_S, U, \alpha, \pi, \rho)$ una interpretación y $\varphi \in L_S$. Definimos recursivamente $I \models \varphi$ como sigue:

1. $I \models Rt_1 \dots t_n \iff (t_1^I, \dots, t_n^I) \in R^I$.
2. $I \models \neg\varphi \iff I \not\models \varphi$, i.e. I no modeliza φ.
3. $I \models \varphi_1 \wedge \varphi_2 \iff I \models \varphi_1$ e $I \models \varphi_2$.
4. $I \models \varphi_1 \vee \varphi_2 \iff I \models \varphi_1$ o $I \models \varphi_2$.

5. $I \models \varphi_1 \to \varphi_2 \iff I \not\models \varphi_1$ o $I \models \varphi_2$.

6. $I \models \varphi_1 \leftrightarrow \varphi_2 \iff$ o bien $I \models \varphi_1$ e $I \models \varphi_2$, o bien $I \not\models \varphi_1$ e $I \not\models \varphi_2$.

7. $I \models \exists x\varphi \iff$ existe un $a \in U$ tal que $I_a^x \models \varphi$.

8. $I \models \forall x\varphi \iff$ para todo $a \in U$, $I_a^x \models \varphi$.

Cuando $I \models \varphi$ diremos que I es un *modelo* de φ o que I *modeliza* φ. Escribiremos $I \not\models \varphi$ cunado I no es un modelo de φ.

Definición 1.21 (Satisfactibilidad)**.** Sean $\varphi \in L_S$ y $\Sigma \subseteq L_S$. Decimos que:

1. φ es *satisfactible* $\iff$ existe una interpretación I tal que $I \models \varphi$.

2. φ es *insatisfactible* $\iff$ φ no es satisfactible.

3. Σ es *satisfactible* $\iff$ existe una interpretación I tal que para toda $\varphi \in \Sigma$, $I \models \varphi$. En este caso decimos que I es un *modelo* de Σ o que I *modeliza* Σ, y lo denotaremos por $I \models \Sigma$.

4. Σ es *insatisfactible* $\iff$ Σ no es satisfactible.

Definición 1.22 (Consecuencia lógica)**.** Sean $\varphi \in L_S$ y $\Sigma \subseteq L_S$. Decimos que φ es *consecuencia lógica* de Σ si para toda interpretación I,

$$I \models \Sigma \implies I \models \varphi.$$

Lo denotaremos por $\Sigma \models \varphi$.

La demostración de la siguiente proposición es inmediata.

Proposición 1.23. *Sean $\varphi \in L_S$ y $\Sigma \subseteq L_S$. Entonces*

$$\varphi \textit{ es consecuencia lógica de } \Sigma \iff \Sigma \cup \{\neg\varphi\} \textit{ es insatisfactible.}$$

1.4. Universo y modelos de Herbrand

Dado un objetivo ψ y un programa lógico P, ver que $P \cup \{\neg\psi\}$ es insatisfactible es, por la proposición 1.23, lo mismo que ver que ψ es consecuencia lógica de P.

Ahora bien, por la definición 1.21, para ver que $P \cup \{\neg\psi\}$ es insatisfactible necesitamos ver que para *TODA* posible interpretación I, $I \not\models P \cup \{\neg\psi\}$. Y eso son muchas interpretaciones que comprobar.
En esta subsección veremos que es suficiente considerar un subconjunto de todas las interpretaciones posibles: las interpretaciones de Herbrand.

Definición 1.24 (Universo de Herbrand)**.** Sea $L = L_S$ un lenguaje de primer orden. Si S no contiene símbolos de constante utilizaremos $S \cup \{c\}$ en su lugar, donde c es un nuevo símbolo de constante. Se define el *universo de Herbrand* U_L de L como el conjunto de términos básicos de T_S, i.e. que se pueden formar con las constantes y los símbolos de función de S.

Definición 1.25 (Base de Herbrand)**.** Sea $L = L_S$ un lenguaje de primer orden. Se define la *base de Herbrand* B_L como el conjunto de todas las fórmulas atómicas cerradas que pueden formarse usando símbolos de relación de S y elementos del universo de Herbrand U_L como argumentos de estos símbolos de relación.

Observación 1.26. Si B_L es base de Herbrand entonces

$$\begin{aligned} B_L =& \{\varphi \in L \colon \varphi \text{ es fórmula atómica cerrada}\} \\ =& At(L) \cap Sen(L). \end{aligned}$$

Definición 1.27 (Interpretación de Herbrand)**.** Sea $L = L_S$ un lenguaje de primer orden. Una *interpretación de Herbrand* H es una interpretación $H = (L, U, \alpha, \pi, \rho)$ de L tal que

1. $U = U_L$, i.e. su universo es el universo de Herbrand de L.
2. Si c es una constante de S, $c^H = c$.
3. Si f es un símbolo de función n–aria de S y $t_1, \ldots, t_n \in U_L$, $f^H t_1 \ldots t_n = f t_1 \ldots t_n \in U_L$.

Observación 1.28. En una interpretación de Herbrand $H = (L, U, \alpha, \pi, \rho)$ tenemos que

- α no juega ningún papel (a partir de esta subsección sólo trataremos con cláusulas, y las cláusulas no contienen variables libres),
- se restringen los posibles universos U y aplicaciones π a una única opción, y
- la definición solamente deja abierta la posibilidad de definir libremente la aplicación ρ.

Tenemos entonces que cada interpretación de Herbrand H queda determinada por el conjunto

$$\{\varphi \in B_L \colon H \models \varphi\} \subseteq B_L,$$

i.e. por el conjunto de fórmulas atómicas cerradas que modeliza.
Por este motivo, de ahora en adelante representaremos una interpretación de Herbrand por el conjunto de fórmulas atómicas cerradas que modeliza, de modo que

- las interpretaciones de Herbrand se convierten en subconjuntos de B_L,
- el vacío $\varnothing$ y B_L son ahora posibles interpretaciones,
- tiene sentido hablar de la unión y de la intersección de interpretaciones, y
- si H es interpretación de Herbrand y $\varphi \in B_L$, tenemos

$$H \models \varphi \iff \varphi \in H.$$

Definición 1.29 (Modelo de Herbrand)**.** Sea L un lenguaje de primer orden y Σ un conjunto de fórmulas cerradas de $Sen(L)$. Un *modelo de Herbrand* para Σ es una interpretación de Herbrand H de L que modeliza Σ, i.e. tal que $H \models \Sigma$.

Veremos ahora que nos basta con considerar las interpretaciones de Herbrand para probar la insatisfactibilidad de un conjunto de *cláusulas*.

Proposición 1.30. *Sea Σ un conjunto finito de cláusulas y supongamos que Σ tiene un modelo. Entonces Σ tiene un modelo de Herbrand.*

Demostración. Sea I un modelo de Σ. Definimos la interpretación de Herbrand

$$H = \{\varphi \in B_L \colon I \models \varphi\}.$$

Veamos que $H \models \Sigma$.
Sea $\forall x_1 \dots \forall x_n \chi$ una cláusula de Σ y sean $t_1, \dots, t_n \in U_L$. Veamos que $H \models \chi[t_1, \dots, t_n]$. Tenemos:

$$\begin{aligned} I \models \Sigma &\implies I \models \forall x_1 \dots \forall x_n \chi \\ &\implies I \models \chi[t_1, \dots, t_n] \\ &\implies I \models \varphi \text{ para algún literal } \varphi \in Mb(\chi[t_1, \dots, t_n]) \\ &\overset{\varphi \in B_L}{\implies} H \models \varphi \text{ para algún literal } \varphi \in Mb(\chi[t_1, \dots, t_n]) \\ &\implies H \models \chi[t_1, \dots, t_n]. \end{aligned}$$

□

Corolario 1.31. *Sea Σ un conjunto finito de cláusulas. Si no existe interpretación de Herbrand que modelice Σ entonces Σ es insatisfactible.*

Tal y como habíamos adelantado al principio de esta subsección, no será necesario comprobar todas las interpretaciones posibles para demostrar la insatisfactibilidad de un conjunto de *cláusulas*, bastará con comprobar el subconjunto de las interpretaciones de Herbrand, que a menudo no sólo será menor sino que incluso será finito.

Observación 1.32. La proposición y el corolario anteriores se han demostrado cuando Σ es un conjunto de *cláusulas*.

No obstante, existe un algoritmo que dada una fórmula cualquiera φ la convierte en una cláusula $\widetilde{\varphi}$ tal que

$$\varphi \text{ es satisfactible} \iff \widetilde{\varphi} \text{ es satisfactible.}$$

Esto nos permitiría tratar con fórmulas que no necesariamente fuesen cláusulas:
Si φ es una fórmula y no existe una interpretación de Herbrand que modelice $\widetilde{\varphi}$ entonces φ es insatisfactible.

En cualquier caso, los programas lógicos están formados únicamente por cláusulas, por lo que exponer dicho algoritmo en este trabajo carece de sentido. El algoritmo involucra el concepto de forma de Skolem, y puede consultarse una versión detallada en [5].

1.5. Modelo mínimo de Herbrand

A partir de ahora trabajaremos con programas lógicos. Por ello, antes de seguir adaptaremos algunas definiciones anteriores para trasladar el énfasis de L_S, un lenguaje de primer orden arbitrario, a P, un programa lógico arbitrario.

La siguiente definición conecta ambos conceptos.

Definición 1.33 (Lenguaje de primer orden inducido por P)**.** Sea P un programa lógico. Definimos el *lenguaje de primer orden* $L^P = L_S$ *inducido por* P como aquél que satisface que las constantes, símbolos de función y símbolos de relación de S son, respectivamente, aquellas constantes, símbolos de función y símbolos de relación que aparecen en alguna cláusula de P.

De esta manera ya podemos definir los conceptos de *universo de Herbrand* U_P y *base de Herbrand* B_P de un programa lógico P.

Definición 1.34 (Universo de Herbrand de P)**.** Sea P un programa lógico. Definimos el *universo de Herbrand* U_P *de* P como el universo de Herbrand definido a partir del lenguaje de primer orden inducido por P, i.e.

$$U_P := U_{L^P}.$$

Definición 1.35 (Base de Herbrand de P)**.** Sea P un programa lógico. Definimos la *base de Herbrand* B_P *de* P como la base de Herbrand definida a partir del lenguaje de primer orden inducido por P, i.e.

$$B_P := B_{L^P}.$$

Dado un programa lógico P pueden existir muchos modelos de Herbrand de P. No obstante hay uno especial: aquél que contiene exactamente los átomos de B_P que son consecuencia lógica de P. Este modelo, que acabaremos llamando *modelo mínimo de Herbrand*, se corresponde con el significado declarativo de un programa lógico, y es el concepto central de la *semántica declarativa*[1] de la Programación Lógica.

Definición 1.36 (Modelo mínimo)**.** Se define el modelo mínimo de Herbrand M_P de un programa P como la intersección de todos los modelos de Herbrand de P.

Observación 1.37. Si el programa P no tuviese modelos tendríamos que

$$M_P = \bigcap \varnothing = \text{ clase universal.}$$

Por suerte esto no pasará nunca ya que todo programa P tiene al menos a B_P como modelo de Herbrand.

[1] A lo largo de este trabajo introduciremos otras semánticas, i.e. otras maneras de interpretar lo que significa un programa lógico, y las compararemos con la semántica declarativa para ver si son o no equivalentes.

Teorema 1.38. *Sea P un programa lógico. Entonces*

$$M_P = \{\varphi \in B_P \colon P \models \varphi\}.$$

Demostración. Sea $\varphi \in B_P$.

$$\begin{aligned} P \models \varphi &\overset{\text{prop 1.23}}{\iff} P \cup \{\neg\varphi\} \text{ es insatisfactible} \\ &\overset{\text{cor 1.31}}{\iff} P \cup \{\neg\varphi\} \text{ no tiene ningún modelo de Herbrand} \\ &\iff \text{si } H \text{ es un modelo de Herbrand de } P, \text{ entonces } H \models \varphi \\ &\iff \varphi \in H \text{ para todo } H \text{ modelo de Herbrand de } P \\ &\iff \varphi \in M_P. \end{aligned}$$

□

1.6. Semántica de punto fijo

En esta subsección introduciremos una nueva semántica, i.e. una nueva manera de interpretar qué significa un programa lógico. Se conoce como la *semántica de punto fijo*, se construye con la ayuda del operador T_P y demostraremos el teorema de van Emden–Kowalski, que afirma que dicha semántica es equivalente a la sémantica declarativa del modelo mínimo de Herbrand.

Necesitaremos un par de resultados de teoría de retículos.

Teorema 1.39 (Tarski)**.** *Si $(A, \leq)$ es un retículo completo y $T \colon A \to A$ es monótona creciente entonces T tiene un mínimo punto fijo* $\operatorname{mpf}(T)$ *tal que*

$$\operatorname{mpf}(T) = \operatorname{ínf}\{b \in A \colon T(b) \leq b\}.$$

Definición 1.40. Sea $(A, \leq)$ un retículo completo y $T \colon A \to A$ monótona creciente. Para todo natural n definimos $T \uparrow n$ por:

- $T \uparrow 0 = \bot$,
- $T \uparrow (n+1) = T(T \uparrow n)$.

Ahora, definimos

$$T \uparrow \omega = \sup\{T \uparrow n : n \geq 0\}.$$

Teorema 1.41 (Kleene)**.** *Si $(A, \leq)$ es un retículo completo y $T \colon A \to A$ es continua entonces*

$$\operatorname{mpf}(T) = T \uparrow \omega.$$

Las demostraciones de estos dos teoremas y las definiciones necesarias para entenderlos aparecen en el apéndice A.

Pasemos ahora a definir el operador T_P.

Notación 1.42. Denotaremos por 2^{B_P} al conjunto de interpretaciones de Herbrand del lenguaje de primer orden L^P.

Observación 1.43. $(2^{B_P}, \subseteq)$ es un retículo completo.

Definición 1.44 (Operador T_P)**.** Sea P un programa lógico. Definimos la aplicación $T_P\colon 2^{B_P} \to 2^{B_P}$ como

$$T_P(I) = \{\varphi \in B_P\colon \text{ existe una realización básica } \varphi \leftarrow \varphi_1, \ldots, \varphi_n \text{ de una cláusula de } P \text{ tal que } \varphi_1, \ldots, \varphi_n \in I\}.$$

Proposición 1.45. *Si P es un programa lógico entonces T_P es continua.*

Demostración. Sea X un subconjunto dirigido de 2^{B_P}. Veamos que

$$\varphi_1, \ldots, \varphi_n \in \sup X \iff \exists I \in X \text{ tal que } \varphi_1, \ldots, \varphi_n \in I. \qquad (*)$$

($\Longleftarrow$) Inmediata.
($\Longrightarrow$) $\varphi_1, \ldots, \varphi_n \in \sup X \implies \varphi_1, \ldots, \varphi_n \in \bigcup\{I\colon I \in X\}$. Sean $I_1, \ldots, I_n \in X$ tales que $\varphi_1 \in I_1, \ldots, \varphi_n \in I_n$. Como X es dirigido, $I_1, \ldots, I_n \subseteq I$ para algún $I \in X$. Por tanto $\varphi_1, \ldots, \varphi_n \in I$.
Veamos ahora que $T_P(\sup X) = \sup T_P[X]$. Sea $\varphi \in B_P$.

$$\begin{aligned}
\varphi \in T_P(\sup X) &\iff \text{existe una realización básica } \varphi \leftarrow \varphi_1, \ldots, \varphi_n \text{ de alguna cláusula de } P \text{ tal que } \varphi_1, \ldots, \varphi_n \in \sup X \\
&\overset{(*)}{\iff} \text{existe una realización básica } \varphi \leftarrow \varphi_1, \ldots, \varphi_n \text{ de alguna cláusula de } P \text{ tal que } \varphi_1, \ldots, \varphi_n \in I \text{ para algún } I \in X \\
&\iff \varphi \in T_P(I) \text{ para algún } I \in X \\
&\iff \varphi \in \bigcup\{T_P(I)\colon I \in X\} \iff \varphi \in \sup T_P[X].
\end{aligned}$$

□

Proposición 1.46. *Si P es un programa lógico e I una interpretación de Herbrand, entonces*

$$I \models P \iff T_P(I) \subseteq I.$$

Demostración. Sea I una interpretación de Herbrand.

$$\begin{aligned}
I \models P &\iff \text{para cada realización básica } \varphi \leftarrow \varphi_1, \ldots, \varphi_n \text{ de cada cláusula de } P, \\
&\qquad \text{si } \varphi_1, \ldots, \varphi_n \in I \text{ entonces } \varphi \in I \\
&\iff \text{si } \varphi \in T_P(I) \text{ entonces } \varphi \in I \\
&\iff T_P(I) \subseteq I.
\end{aligned}$$

□

Sabiendo todo esto podemos enunciar y demostrar el siugiente teorema.

Teorema 1.47 (Van Emden–Kowalski)**.** *Si P es un programa lógico entonces*

$$M_P = \mathrm{mpf}(T_P) = T_P \uparrow \omega.$$

Demostración.

$$\begin{aligned} M_P &= \bigcap\{I \colon I \text{ modelo de Herbrand de } P\} = \bigcap\{I \in 2^{B_P} \colon I \models P\} \\ &= \text{ínf}\{I \in 2^{B_P} \colon I \models P\}, \text{ por ser el orden la inclusión} \\ &= \text{ínf}\{I \colon T_P(I) \subseteq I\}, \text{ por la proposición 1.46} \\ &= \mathrm{mpf}(T_P), \text{ por el teorema 1.39} \\ &= T_P \uparrow \omega, \text{ por el teorema 1.41 y la proposición 1.45.} \end{aligned}$$

□

2. Semántica procedural: la SLD–Resolución

En esta sección, nuestro objetivo es demostrar la corrección y la completud de la SLD–Resolución, un refinamiento de la Resolución de Robinson. En primer lugar necesitaremos introducir algunos conceptos.

2.1. Unificación

El propósito de esta subsección es describir el algoritmo de unificación, pieza clave de la SLD–Resolución[2].

Definición 2.1 (Sustitución)**.** Una *sustitución* θ es un conjunto finito

$$\{v_1/t_1, \ldots, v_n/t_n\}$$

donde todas las v_i son variables distintas y cada t_i es un término distinto de v_i.
Si todos los t_i son términos básicos decimos que la sustitución es *básica*.
Llamamos a la sustitución $\varnothing$ la *sustitución vacía* y la denotaremos por ε.

Definición 2.2 (Expresión)**.** Una *expresión* es o bien un término, o bien un literal, o bien una conjunción o disyunción de literales.
Una *expresión simple* es o bien un término o bien una fórmula atómica, i.e. una expresión sin conectivas.

Definición 2.3 (Realización)**.** Sean $\theta = \{v_1/t_1, \ldots, v_n/t_n\}$ una sustitución y E una *expresión*. Definimos la *realización* $E\theta$ de E por θ como la expresión obtenida al reemplazar cada ocurrencia de la variable v_i por el término t_i para todo $i = 1, \ldots, n$.
Si $E\theta$ es básica, decimos que la realización es *básica*.
Si $\varphi = \forall(E)$ es una *cláusula de programa* y θ es una sustitución, definimos la realización de φ por θ como $\forall(E\theta)$. Si $\forall(E\theta) = E\theta$, decimos que la realización es *básica*.
Si $\psi = \exists(E)$ es un *objetivo* y θ es una sustitución, definimos la realización de ψ por θ como $\psi\theta := \exists(E\theta)$.

Definición 2.4 (Composición de sustituciones)**.** Sean $\theta = \{u_1/s_1, \ldots, u_m/s_m\}$ y $\sigma = \{v_1/t_1, \ldots, v_n/t_n\}$ dos sustituciones. Definimos la *composición* $\theta\sigma$ de las sustituciones θ y σ como

$$\theta\sigma = \{u_i/s_i\sigma \colon u_i \neq s_i\sigma\} \cup \{v_i/t_i \in \sigma \colon v_i \neq u_j \forall j\}.$$

Notación 2.5. Si $W = \{E_1, \ldots, E_n\}$ es un conjunto finito de expresiones y θ es una sustitución entonces $W\theta$ denota el conjunto $\{E_1\theta, \ldots, E_n\theta\}$.

Definición 2.6 (Unificador)**.** Sean E y F dos expresiones y W un conjunto finito de expresiones.

[2] y en general de cualquier variante de la Resolución de Robinson.

- Una sustitución θ es un *unificador* de E y F si $\{E, F\}\theta$ es un conjunto unitario. En general, una sustitución θ es un *unificador* de W si $W\theta$ es un conjunto unitario.

- E y F *unifican* si existe un unificador de $\{E, F\}$. En general, W *unifica* si existe un unificador de W.

- Un unificador θ de W es un *unificador de máxima generalidad* (u.m.g.) de W si para todo unificador σ de W existe una sustitución γ tal que $\sigma = \theta\gamma$.

Definimos a continuación un algoritmo que dado un conjunto finito de expresiones simples W nos dice si W unifica y en caso afirmativo nos devuelve un u.m.g. de W.

Definición 2.7 (Conjunto de desacuerdo)**.** Sea W un conjunto finito de expresiones simples. Definimos el conjunto de desacuerdo de W como el conjunto de las subexpresiones de las expresiones de W formadas a partir de la primera posición donde las expresiones de W no coinciden.

Ejemplo 2.8. Sea $W = \big\{P\big(f\big(f(x)\big), g(x, y)\big), P\big(f\big(g(c, d)\big), g(c, x)\big)\big\}$. Entonces el conjunto de desacuerdo de W es $\{f(x), g(c, d)\}$.

Algoritmo 2.9 (de unificación)**.** El algoritmo de unificación recibe como entrada un conjunto $W \neq \varnothing$ finito de expresiones simples. El algoritmo va generando conjuntos de expresiones simples $W_0 = W, W_1, \ldots, W_k, \ldots$ y sustituciones $\sigma_0 = \varepsilon, \sigma_1, \ldots, \sigma_k, \ldots$.

- PASO 1: $k := 0$, $W_0 := W$, $\sigma_0 := \varepsilon$.

- PASO 2: Si W_k es un conjunto unitario, STOP 1. Si no, se construye el conjunto de desacuerdo D_k de W_k.

- PASO 3: Si existen $v_k, t_k \in D_k$ tales que v_k es una variable que no aparece en el término t_k entonces $\sigma_{k+1} := \sigma_k\{v_k/t_k\}$, $W_{k+1} := W_k\{v_k/t_k\}$, $k := k + 1$ y se vuelve al PASO 2. En otro caso, STOP 2.

Teorema 2.10 (de unificación)**.** *Sea W un conjunto finito de expresiones simples. Si W unifica entonces el algoritmo de unificación termina en STOP 1 para algún k y σ_k es un u.m.g. de W. Si W no unifica, el algoritmo termina en STOP 2.*

Demostración. Primero, el algoritmo siempre termina: W contiene una cantidad finita de variables y cada aplicación del PASO 3 elimina una variable.

Segundo, basta ver que cuando W unifica el algoritmo de unificación termina en STOP 1 y nos devuelve un u.m.g. de W. En efecto, en tal caso si W no unifica el algoritmo (que siempre termina) sólo puede terminar en STOP 2.

Veamos pues que si W unifica entonces el algoritmo de unificación devuelve un u.m.g. de W. Supongamos que W unifica y sea θ un unificador de W. Probemos por inducción que para todo $k \geq 0$, si σ_k es la k–ésima sustitución generada por el algoritmo entonces $\theta = \sigma_k\gamma_k$ para alguna sustitución γ_k:

- Si $k = 0$, tenemos que $\gamma_0 = \theta$ satisface $\theta = \varepsilon\theta$.
- Si $k > 0$, supongamos como hipótesis de inducción el resultado para $k-1$. Si el algoritmo ha calculado σ_k es porque W_{k-1} no era un conjunto unitario, por lo que el algoritmo calculó el conjunto de desacuerdo D_{k-1} de W_{k-1} y pasó al PASO 3. Por hipótesis de inducción $\theta = \sigma_{k-1}\gamma_{k-1}$, de modo que $W\theta = W\sigma_{k-1}\gamma_{k-1}$. Como θ unifica W, necesariamente γ_{k-1} debe unificar $W\sigma_{k-1} = W_{k-1}$, y en particular a $\{v_{k-1}, t_{k-1}\} \in W_{k-1}$, con v_{k-1} y t_{k-1} la variable y el término que escogió el algoritmo en la iteración $k-1$. Entonces $v_{k-1}\gamma_{k-1} = t_{k-1}\gamma_{k-1}$. Tenemos pues que $\sigma_k = \sigma_{k-1}\{v_{k-1}/t_{k-1}\}$. Sea $\gamma_k = \gamma_{k-1} \setminus \{v_{k-1}/v_{k-1}\gamma_{k-1}\}$. Entonces

$$\begin{aligned}\{v_{k-1}/t_{k-1}\}\gamma_k &= \{v_{k-1}/t_{k-1}\}(\gamma_{k-1} \setminus \{v_{k-1}/v_{k-1}\gamma_{k-1}\}) = \\ &= \{v_{k-1}/t_{k-1}(\gamma_{k-1} \setminus \{v_{k-1}/v_{k-1}\gamma_{k-1}\})\} \cup \{(\gamma_{k-1} \setminus \{v_{k-1}/v_{k-1}\gamma_{k-1}\}) = \\ &= \{v_{k-1}/t_{k-1}\gamma_{k-1}\} \cup (\gamma_{k-1} \setminus \{v_{k-1}/v_{k-1}\gamma_{k-1}\}) = \gamma_{k-1}.\end{aligned}$$

  Por la asociatividad de las sustituciones y por la hipótesis de inducción tenemos

$$\sigma_k\gamma_k = (\sigma_{k-1}\{v_{k-1}/t_{k-1}\})\gamma_k = \sigma_{k-1}(\{v_{k-1}/t_{k-1}\}\gamma_k) = \sigma_{k-1}\gamma_{k-1} = \theta.$$

□

2.2. SLD–Resolución

Hay muchos procesos de refutación basados en la Resolución de Robinson. La SLD–Resolución (Selection function–Linear–Definite Clauses–Resolution) es uno de ellos y fue descrita por primera vez por Kowalski. En esta subsección describiremos en qué consiste.

Notación 2.11. Si $\varphi = \varphi_0 \leftarrow \varphi_1, \ldots, \varphi_n$ es una cláusula de programa entonces definimos

$$\begin{aligned}\varphi^+ &:= \varphi_0, \\ \varphi^- &:= (\varphi_1 \wedge \cdots \wedge \varphi_n).\end{aligned}$$

Definición 2.12 (Regla de computación)**.** Una *regla de computación* es una aplicación R que a cada objetivo le asigna una de sus fórmulas atómicas, que llamamos la *fórmula atómica elegida*.

Definición 2.13 (Resolvente)**.** Sean $\psi = \exists(\chi_1 \wedge \cdots \wedge \chi_m \wedge \cdots \wedge \chi_k)$ un objetivo, $\varphi = \varphi^+ \leftarrow \varphi^-$ una cláusula de programa y R una regla de computación. Supongamos que $R(\psi) = \chi_m$ y que θ es un unificador de χ_m y φ^+. Diremos entonces que el objetivo

$$\psi' = \exists(\chi_1 \wedge \cdots \wedge \chi_{m-1} \wedge \varphi^- \wedge \chi_{m+1} \wedge \cdots \wedge \chi_k)\theta$$

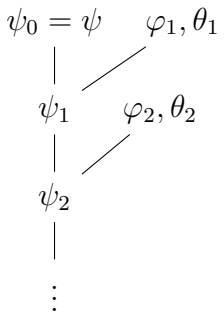
es un *resolvente* de ψ y φ a partir de θ vía R.

Definición 2.14 (Variante)**.** Sean E y F expresiones. Decimos que E es una *variante* de F si existen sustituciones θ y σ tales que $E = F\theta$ y $F = E\sigma$.

Definición 2.15 (SLD–derivación)**.** Sean P un programa lógico, ψ un objetivo y R una regla de computación. Una *SLD–derivación* de $P \cup \{\neg\psi\}$ vía R está compuesta por tres sucesiones asociadas:

- una sucesión $(\psi_0 = \psi, \psi_1, \dots)$ de objetivos,
- una sucesión $(\varphi_1, \varphi_2, \dots)$ de variantes de cláusulas de P y
- una sucesión $(\theta_1, \theta_2, \dots)$ de u.m.g.'s,

de modo que cada ψ_{i+1} es un resolvente de ψ_i y φ_{i+1} a partir de θ_{i+1} vía R. Representaremos una tal SLD–derivación mediante el siguiente árbol:



Llamaremos a dicho árbol el *árbol de derivación* de la SLD–derivación.

Observación 2.16. Asumiremos que en una SLD–derivación las variantes de la sucesión asociada $(\varphi_1, \varphi_2, \dots)$ cumplen que

- $\mathrm{Var}(\varphi_i) \cap \mathrm{Var}(\psi_j) = \varnothing, \quad \forall i,\ \forall j < i,$
- $\mathrm{Var}(\varphi_i) \cap \mathrm{Var}(\varphi_j) = \varnothing, \quad \forall i \neq j.$

Esto es porque dada una SLD–derivación siempre se pueden renombrar las variables de todas las variantes φ_i de manera que se cumplan las condiciones. El proceso se denomina "estandarizar las variables".

Definición 2.17 (Clasificación)**.** Una SLD–derivación puede ser

- o bien *infinita*,
- o bien *de éxito*, si es finita y el último objetivo es la cláusula vacía $\square$,
- o bien *de fallo*, si es finita, el último objetivo no es $\square$ y la última fórmula atómica elegida no es unificable con ninguna cabeza de las cláusulas de P,
- o bien *incompleta*, en otro caso.

Si una SLD–derivación es finita y su último objetivo es ψ_n diremos que la *longitud* de la SLD–derivación es n.

Definición 2.18 (SLD–refutación)**.** Llamaremos *SLD–refutación* de $P \cup \{\neg\psi\}$ vía R a una SLD–derivación de éxito de $P \cup \{\neg\psi\}$ vía R.

2.3. Corrección de la SLD–Resolución

Definición 2.19 (Respuesta)**.** Sean P un programa lógico y ψ un objetivo. Una *respuesta* para $P \cup \{\neg\psi\}$ es una sustitución θ para algunas variables de ψ.

Definición 2.20 (Respuesta correcta)**.** Sean P un programa lógico, ψ un objetivo $\exists(\chi_1 \wedge \cdots \wedge \chi_n)$ y θ una respuesta para $P \cup \{\neg\psi\}$. Decimos que θ es una *respuesta correcta* para $P \cup \{\neg\psi\}$ si $\forall((\chi_1 \wedge \cdots \wedge \chi_k)\theta)$ es consecuencia lógica de P.

Para probar la corrección y la completud de los programas lógicos, demostraremos la equivalencia entre el concepto declarativo de respuesta correcta y su correspondiente concepto procedural.

En la siguiente proposición, mostraremos una propiedad básica de las respuestas correctas.

Proposición 2.21. *Sean P un programa lógico, $\psi = \exists(\chi_1 \wedge \cdots \wedge \chi_k)$ un objetivo y θ una respuesta para $P \cup \{\neg\psi\}$ tal que $\psi\theta$ es una fórmula básica. Entonces son equivalentes las siguientes condiciones:*

1. *θ es correcta.*
2. *$M \models \psi\theta$ para todo M modelo de Herbrand de P.*
3. *$M_P \models \psi\theta$.*

Demostración.

$$
\begin{aligned}
\theta \text{ correcta} &\overset{\text{def 2.20}}{\iff} \forall((\chi_1 \wedge \cdots \wedge \chi_n)\theta) \text{ es consecuencia lógica de } P \\
&\iff \forall(\chi_1\theta), \ldots, \forall(\chi_n\theta) \text{ son consecuencia lógica de } P \\
&\overset{\text{teo 1.38}}{\iff} \chi_1\theta, \ldots, \chi_n\theta \in M_P, \text{ por ser } \psi\theta \text{ fórmula básica} \\
&\overset{\text{obs 1.28}}{\iff} M_P \models \chi_i\theta \quad \forall i = 1, \ldots, n \\
&\iff M_P \models \psi\theta \\
&\iff M \models \psi\theta \text{ para todo } M \text{ modelo de Herbrand de } P.
\end{aligned}
$$

□

Definimos el análogo procedural del modelo mínimo de Herbrand.

Definición 2.22 (Conjunto de éxito)**.** Sea P un programa lógico. Definimos el *conjunto de éxito* E_P de P como

$$E_P = \{\varphi \in B_P \colon P \cup \{\neg\varphi\} \text{ tiene una SLD–refutación}\}.$$

Introducimos ahora la contraparte procedural del concepto de respuesta correcta.

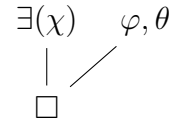
Definición 2.23 (R–respuesta computada)**.** Sean P un programa lógico, ψ un objetivo y R regla de computación. Una *R–respuesta computada* θ para $P \cup \{\neg\psi\}$ es la sustitución que se obtiene al restringir la composición $\theta_1 \dots \theta_n$ a las variables de ψ, donde $(\theta_1, \dots, \theta_n)$ es la sucesión de u.m.g.'s de una SLD–refutación de $P \cup \{\neg\psi\}$ vía R.
Una *respuesta computada* para $P \cup \{\neg\psi\}$ es una R–respuesta computada para $P \cup \{\neg\psi\}$ para alguna regla de computación R.

Demostraremos ahora la corrección de la SLD-resolución viendo que toda R–respuesta computada es correcta.

Teorema 2.24 (de corrección de Clark)**.** *Sean P un programa lógico, ψ un objetivo y R una regla de computación. Entonces toda R–respuesta computada para $P \cup \{\neg\psi\}$ es una respuesta correcta para $P \cup \{\neg\psi\}$.*

Demostración. Sea ψ el objetivo $\exists(\chi_1 \wedge \dots \wedge \chi_k)$ y $(\theta_1, \dots, \theta_n)$ la sucesión de u.m.g.'s asociados a una SLD–refutación de $P \cup \{\neg\psi\}$ vía R.
Debemos ver que $\forall((\chi_1 \wedge \dots \wedge \chi_k)\theta_1 \dots \theta_n)$ es consecuencia lógica de P. Lo hacemos por inducción sobre la longitud n de la SLD–refutación.

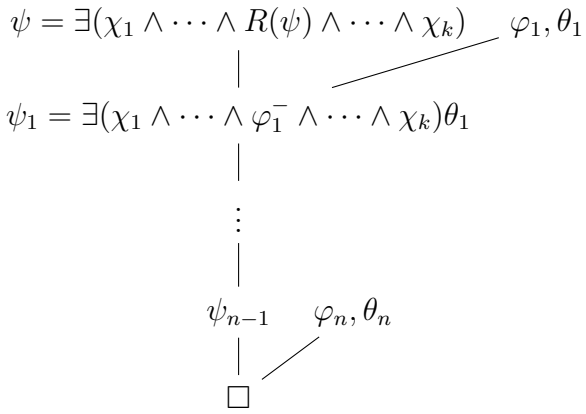
- CASO $n = 1$. La SLD–refutación será de la forma



  Entonces $\varphi = \forall(\varphi)$ es un hecho de P y $\chi\theta = \varphi\theta$, de modo que

$$\varphi \in P \implies P \models \varphi \implies P \models \varphi\theta = \forall(\varphi\theta) = \forall(\chi\theta).$$

- CASO $n > 1$. La SLD–refutación será de la forma



  Entonces ψ_1 tiene una SLD–refutación de longitud n, y por la hipótesis de inducción

$$P \models \forall((\chi_1 \wedge \dots \wedge \varphi_1^- \wedge \dots \wedge \chi_k)\theta_1\theta_2 \dots \theta_n) \qquad (1)$$

En particular

$$P \models \forall((\varphi_1^-)\theta_1\theta_2 \dots \theta_n). \tag{2}$$

Por otro lado φ_1 es variante de una cláusula φ_1' de P, i.e. existe σ sustitución tal que $\varphi_1 = \varphi_1'\sigma$. Entonces

$$\varphi_1' \in P \implies P \models \varphi_1' \implies P \models \varphi_1'\sigma = \varphi_1 = \forall(\varphi_1^- \to \varphi_1^+). \tag{3}$$

Entonces por (2) y (3) tenemos

$$P \models \forall((\varphi_1^+)\theta_1\theta_2 \dots \theta_n).$$

Del árbol de derivación deducimos que $R(\psi)\theta_1 = \varphi_1^+\theta_1$, de modo que

$$P \models \forall(R(\psi)\theta_1\theta_2 \dots \theta_n). \tag{4}$$

Entonces por (1) y (4) tenemos

$$P \models \forall((\chi_1 \wedge \dots \wedge R(\psi) \wedge \dots \wedge \chi_k)\theta_1 \dots \theta_n).$$

□

Corolario 2.25. *El conjunto de éxito de un programa lógico P está contenido en su modelo mínimo de Herbrand, i.e.*

$$E_P \subseteq M_P.$$

Demostración. Sea P un programa lógico, $\psi \in B_P$ y supongamos que $P \cup \{\neg\psi\}$ tiene una SLD–refutación. Entonces $P \cup \{\neg\psi\}$ tiene una respuesta computada que por el teorema 2.24 es una respuesta correcta, i.e. $\forall(\psi\theta)$ es consecuencia de P. Pero ψ no tiene variables libres porque $\psi \in B_P$, por lo que $\forall(\psi\theta) = \psi\theta = \psi$, de modo que $\psi \in M_P$. Entonces

$$E_P = \{\psi \in B_P \colon P \cup \{\neg\psi\} \text{ tiene una SLD–refutación}\} \subseteq \{\psi \in B_P \colon P \models \psi\} = M_P.$$

□

2.4. Completud de la SLD–Resolución

En esta subsección veremos los teoremas de completud de Apt–van Emden, de Hill y de Clark.

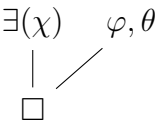
Para ver el primero comenzamos enunciando y demostrando dos lemas que nos serán útiles. Pero antes una definición.

Definición 2.26 (SLD–refutación débil)**.** Una *SLD–refutación débil* es igual que una SLD–refutación, salvo que las sustituciones θ_i de la sucesión asociada $(\theta_1, \dots, \theta_n)$ son unificadores no necesariamente de máxima generalidad.

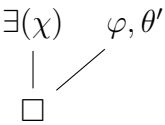
Lema 2.27. *Sean P un programa lógico y ψ un objetivo. Supongamos que $P \cup \{\neg\psi\}$ tiene una SLD–refutación débil. Entonces $P \cup \{\neg\psi\}$ tiene una SLD–refutación de la misma longitud.*
Además, si $(\theta_1, \ldots, \theta_n)$ y $(\theta'_1, \ldots, \theta'_n)$ son las sucesiones de unificadores asociadas a la SLD–refutación débil y a la SLD–refutación respectivamente, entonces existe una sustitución γ tal que $\theta_1 \ldots \theta_n = \theta'_1 \ldots \theta'_n \gamma$.

Demostración. Por inducción sobre la longitud n de la SLD–refutación débil.

- CASO $n = 1$. Tenemos

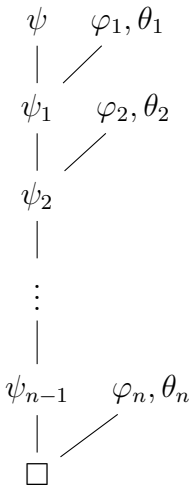


  y $\chi\theta = \varphi\theta$. Por el teorema de unificación 2.10, existe un u.m.g. θ' de χ y φ. Tenemos entonces

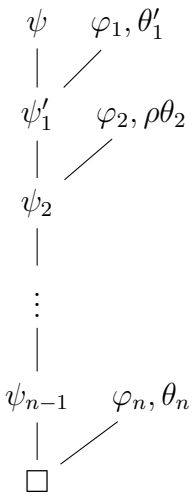


  y $\theta = \theta'\gamma$ para alguna sustitucion γ, ya que θ' es de máxima generalidad.

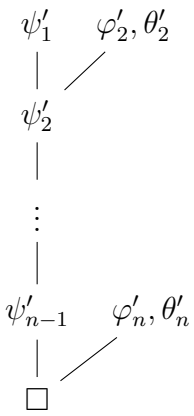
- CASO $n > 1$. Supongamos que $P \cup \{\neg\psi\}$ tiene una SLD–refutación débil de longitud n de la forma:



  Como θ_1 es un unificador de φ_1^+ y $R(\psi)$, por el teorema de unificación 2.10, existe un u.m.g. θ'_1 de φ_1^+ y $R(\psi)$. Por tanto, existe una sustitución ρ tal que $\theta_1 = \theta'_1\rho$. Tenemos entonces la siguiente SLD–refutación:

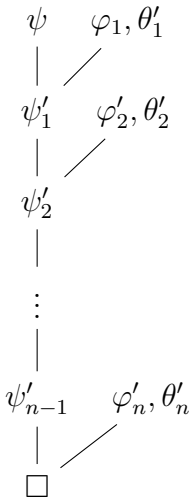


Por la hipótesis de inducción, existe una SLD–refutación para $P \cup \{\neg\psi'_1\}$ de la forma



tal que $\rho\theta_2\theta_3 \ldots \theta_n = \theta'_2 \ldots \theta'_n\gamma$ para alguna sustitución γ.

Por tanto tenemos

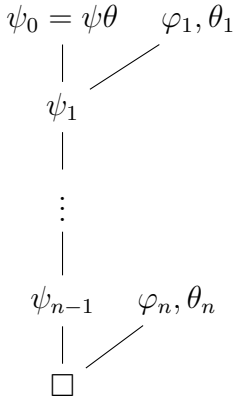


y $\theta_1 \dots \theta_n = \theta'_1 \rho\theta_2 \dots \theta_n = \theta'_1\theta'_2 \dots \theta'_n\gamma$.

□

Lema 2.28 (de ascensión)**.** *Sean P un programa lógico, ψ un objetivo y θ una sustitución. Supongamos que existe una SLD–refutación de $P \cup \{\neg\psi\theta\}$ con u.m.g.'s $\theta_1, \dots, \theta_n$. Entonces existe una SLD–refutación de $P \cup \{\neg\psi\}$ de la misma longitud con u.m.g.'s $\theta'_1, \dots, \theta'_n$ de modo que existe una sustitución γ tal que $\theta\theta_1 \dots \theta_n = \theta'_1 \dots \theta'_n\gamma$.*

Demostración. Supongamos que $P \cup \{\neg\psi\theta\}$ tiene una SLD–refutación de la forma

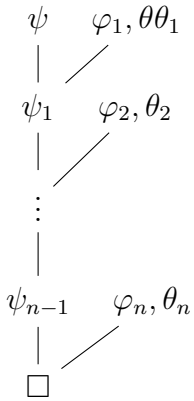


Entonces $R(\psi_0)\theta_1 = \varphi_1^+\theta_1$. Claramente $R(\psi_0) = R(\psi\theta) = R(\psi)\theta$ y, como φ_1 es una variante de una cláusula de P, podemos suponer sin pérdida de generalidad que θ no afecta a las variables de φ_1, i.e. $\varphi_1^+ = \varphi_1^+\theta$, de manera que

$$R(\psi)\theta\theta_1 = \varphi_1^+\theta\theta_1,$$

i.e. $\theta\theta_1$ es unificador de φ_1^+ y $R(\psi)$.

Entonces, la derivación



es una SLD–refutación débil de $P \cup \{\neg\psi\}$, y por el lema 2.27 existe una SLD–refutación de $P \cup \{\neg\psi\}$ de la misma longitud con sucesión de u.m.g.'s $(\theta'_1, \dots, \theta'_n)$ tal que existe una sustitución γ que satisface

$$\theta\theta_1 \dots \theta_n = \theta'_1 \dots \theta'_n\gamma.$$

□

Este primer teorema es el recíproco del corolario 2.25.

Teorema 2.29 (de completud de Apt–van Emden)**.** *El modelo mínimo de Herbrand de un programa lógico P está contenido en su conjunto de éxito, i.e.*

$$M_P \subseteq E_P.$$

Demostración. Tenemos que ver que

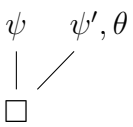
$$M_P = \{\varphi \in B_P \colon P \models \varphi\} \subseteq \{\varphi \in B_P \colon P \cup \{\neg\varphi\} \text{ tiene una SLD–refutación}\} = E_P.$$

Sea $\psi \in M_P$, i.e. $\psi \in B_P$ tal que $P \models \psi$. Por el teorema 1.47 sabemos que $M_P = T \uparrow \omega$, de manera que $\psi \in T \uparrow n$ para algún $n \in \omega$.
Veamos por inducción sobre n que entonces $P \cup \{\neg\psi\}$ tiene una SLD–refutación.

- CASO $n = 1$.

  $$\psi \in T_P \uparrow 1 = T_P(\varnothing) = \{\varphi \in B_P \colon \varphi \text{ es una realización básica de un hecho } \varphi' \text{ de } P\}.$$

  Sea entonces ψ' el hecho de P del cual ψ es una realización básica. En tal caso

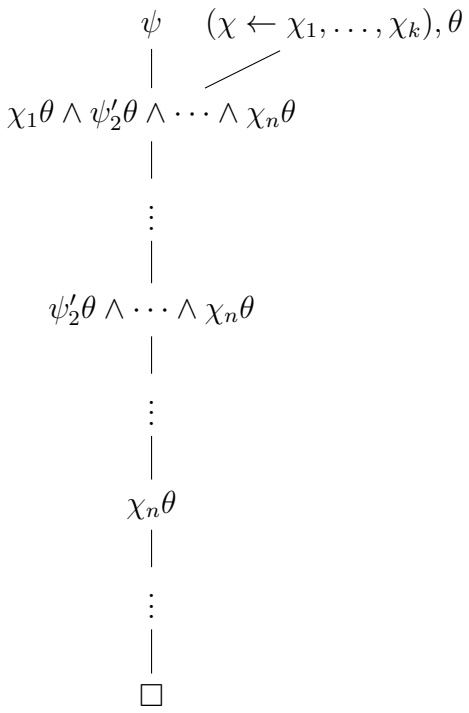
  donde θ es un u.m.g. de ψ y ψ' (que existe por el teorema de unificación 2.10), es una SLD–refutación de $P \cup \{\neg\psi\}$.

- CASO $n > 1$.

  $$\psi \in T_P \uparrow n = \{\varphi \in B_P \colon \text{existe una realización básica } \varphi \leftarrow \varphi_1, \ldots, \varphi_k \text{ de una cláusula de } P \text{ tal que } \varphi_1, \ldots, \varphi_k \in T_P \uparrow (n-1)\}.$$

  Entonces existe una cláusula de P, $\chi \leftarrow \chi_1, \ldots, \chi_k$, y una sustitución θ tal que $\psi = \chi\theta$ y $\chi_1\theta, \ldots, \chi_k\theta \in T_P \uparrow (n-1)$.

  Por la hipótesis de inducción, $P \cup \{\neg\chi_1\theta\}, \ldots, P \cup \{\neg\chi_n\theta\}$ tienen una SLD–refutación. Como $\neg\chi_1\theta, \ldots, \neg\chi_n\theta$ son básicas, podemos combinar estas SLD–refutaciones para obtener una refutación débil de $P \cup \{\neg\psi\}$:



Y por el lema 2.27, $P \cup \{\neg\psi\}$ tiene una SLD–refutación.

$\square$

Veamos el segundo teorema de completud.

Teorema 2.30 (de completud de Hill)**.** *Sean P un programa lógico y ψ un objetivo. Supongamos que $P \cup \{\neg\psi\}$ es insatisfactible. Entonces existe una regla de computación R y una SLD–refutación de $P \cup \{\neg\psi\}$ vía R.*

Demostración. $P \cup \{\neg\psi\}$ es insatisfactible $\implies$ $M_P \not\models \forall(\neg\psi)$ $\implies$ existe una realización básica $\psi\theta$ de ψ tal que $M_P \models \psi\theta$ $\implies$ $M_P \models \psi_1\theta, \ldots, \psi_n\theta \overset{\text{teo 2.29}}{\implies}$ $\psi_1\theta, \ldots, \psi_n\theta \in E_P$ $\implies$ existen SLD–refutaciones de $P \cup \{\neg\psi_1\theta\}, \ldots, P \cup \{\neg\psi_n\theta\}$.

Como cada $\neg\psi_i\theta$ es una fórmula básica, podemos combinar estas SLD–refutaciones para obtener una SLD–refutación de $P \cup \{\neg\psi\theta\}$. Aplicando ahora el lema de ascensión 2.28, obtenemos una SLD–refutación para $P \cup \{\neg\psi\}$. $\square$

Nos fijamos finalmente en las respuestas correctas. No es posible probar el recíproco del teorema de corrección de Clark 2.24 porque las respuestas computadas son siempre "de máxima generalidad". Veremos en cambio que cada respuesta correcta θ es la "realización" de una respuesta computada σ, i.e. que existe una sustitución γ tal que $\theta = \sigma\gamma$. Primero, necesitamos demostrar el siguiente lema.

Lema 2.31. *Sean P un programa lógico y ψ una fórmula atómica. Supongamos que $\forall(\psi)$ es consecuencia lógica de P. Entonces existe una SLD–refutación de $P \cup \{\neg\exists(\psi)\}$ cuya respuesta computada es la sustitución vacía ε.*

Demostración. Sean $x_1, \ldots, x_n$ las variables de ψ. Escogemos $a_1, \ldots, a_n$ constantes diferentes y que no aparezcan ni en P ni en ψ, y definimos $\theta = \{x_1/a_1, \ldots, x_n/a_n\}$. Como $P \models \forall(\psi)$ entonces $P \models \psi\theta$. Como $\psi\theta$ es una realización básica de $\forall(\psi)$, por el teorema de completud de Apt–van Emden 2.29, $P \cup \{\neg\psi\theta\}$ tiene una SLD–refutación cuya respuesta computada será la sustitución vacía ε. Si en dicha SLD–refutación sustituimos cada aparición de una constante a_i por la variable x_i entonces obtenemos una SLD–refutación de $P \cup \{\neg\exists(\psi)\}$ cuya respuesta computada sigue siendo ε. □

Teorema 2.32 (de completud de Clark)**.** *Sean P un programa lógico y ψ un objetivo. Para toda respuesta correcta θ para $P \cup \{\neg\psi\}$, existen una regla de computación R, una R–respuesta computada σ para $P \cup \{\neg\psi\}$ y una sustitución γ tal que $\theta = \sigma\gamma$.*

Demostración. Sea $\psi = \exists(\psi_1 \wedge \cdots \wedge \psi_k)$. Como θ es correcta, $\forall((\psi_1 \wedge \cdots \wedge \psi_k)\theta)$ es consecuencia lógica de P, i.e. $\forall(\psi_i\theta)$ es consecuencia lógica de P para todo $i = 1, \ldots, k$. Por el lema 2.31, existe una SLD–refutación de $P \cup \{\neg\psi_i\theta\}$ con ε como respuesta computada para todo $i = 1, \ldots, k$, y combinando dichas SLD–refutaciones obtenemos una SLD–refutación de $P \cup \{\neg\psi\theta\}$ con ε como respuesta computada. Supongamos que $(\theta_1, \ldots, \theta_n)$ es la sucesión de u.m.g.'s asociada a dicha SLD–refutación de $P \cup \{\neg\psi\theta\}$. Como la respuesta computada es ε,

$$\psi\theta = \psi\theta\theta_1 \ldots \theta_n.$$

Por el lema de ascensión 2.28, existe una SLD–refutación de $P \cup \{\neg\psi\}$ con sucesión de u.m.g.'s $(\theta'_1, \ldots, \theta'_n)$ tal que existe una sustitución γ' que satisface

$$\theta\theta_1 \ldots \theta_n = \theta'_1 \ldots \theta'_n\gamma'.$$

Sea σ la respuesta computada de dicha SLD–refutación de $P \cup \{\neg\psi\}$. Tenemos que

$$\psi\sigma = \psi\theta'_1 \ldots \theta'_n.$$

Entonces

$$\psi\theta = \psi\theta\theta_1 \ldots \theta_n = \psi\theta'_1 \ldots \theta'_n\gamma',$$

i.e. que θ, que ya estaba restringida a las variables de ψ por ser respuesta, es igual a $\theta'_1 \ldots \theta'_n\gamma'$ restringida a las variables de ψ. Pero $\theta'_1 \ldots \theta'_n$ restringida a las variables de ψ es la respuesta computada σ.

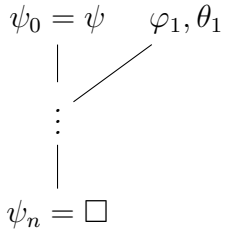
Pongamos que $\sigma = \{x_1/t_1, \ldots, x_k/t_k\}$, y sea γ la restricción de γ' a las variables qua aparecen en $t_1, \ldots, t_k$. Entonces $\theta = \sigma\gamma$. □

2.5. Independencia de las reglas de computación

Veamos ahora el teorema de independencia de las reglas de computación. Por el teorema de completud de Hill 2.30, si $P \cup \{\neg\psi\}$ es insatisfactible existe una SLD–refutación de $P \cup \{\neg\psi\}$. Lo que nos dirá el teorema de independencia es que, en realidad, para toda regla de computación R existe una SLD–refutación de $P \cup \{\neg\psi\}$ vía R.

Para demostrar el teorema necesitaremos un lema técnico.

Lema 2.33 (de intercambio)**.** *Sean P un programa lógico, ψ un objetivo y R una regla de computación. Supongamos que $P \cup \{\neg\psi\}$ tiene una SLD–refutación vía R*



Fijemos q tal que $0 < q < n$. Pongamos $R(\psi_{q-1}) = \chi_i$, $R(\psi_q) = \chi_j$,

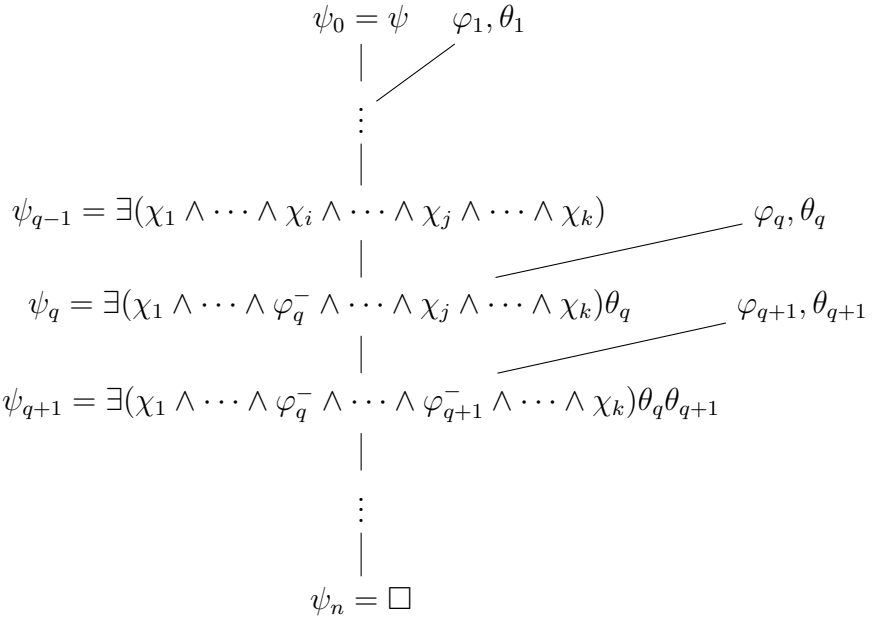
$$\begin{aligned}\psi_{q-1} &= \exists(\chi_1 \wedge \cdots \wedge \chi_i \wedge \cdots \wedge \chi_j \wedge \cdots \wedge \chi_k),\\ \psi_q &= \exists(\chi_1 \wedge \cdots \wedge \varphi_q^- \wedge \cdots \wedge \chi_j \wedge \cdots \wedge \chi_k)\theta_q,\\ \psi_{q+1} &= \exists(\chi_1 \wedge \cdots \wedge \varphi_q^- \wedge \cdots \wedge \varphi_{q+1}^- \wedge \cdots \wedge \chi_k)\theta_q\theta_{q+1}.\end{aligned}$$

Entonces existe una SLD–refutación de $P \cup \{\neg\psi\}$ vía R', donde R' es una regla de computación igual a R salvo en

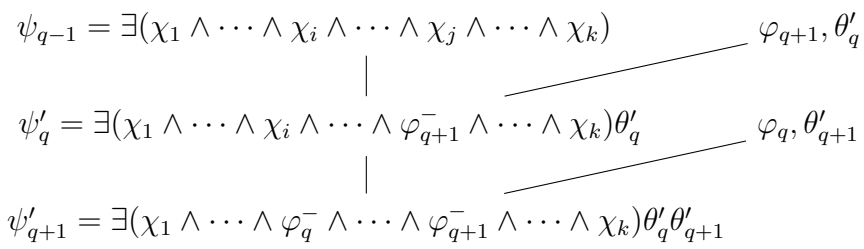
$$R'(\psi_{q-1}) = \chi_j, \text{ y } R'(\psi_q) = \chi_i\theta'_q,$$

con θ'_q u.m.g. de φ_{q+1}^+ y χ_j. Además, si σ es la R–respuesta computada para $P \cup \{\neg\psi\}$ y σ' es la R'–respuesta computada para $P \cup \{\neg\psi\}$, entonces $\psi\sigma$ es una variante de $\psi\sigma'$.

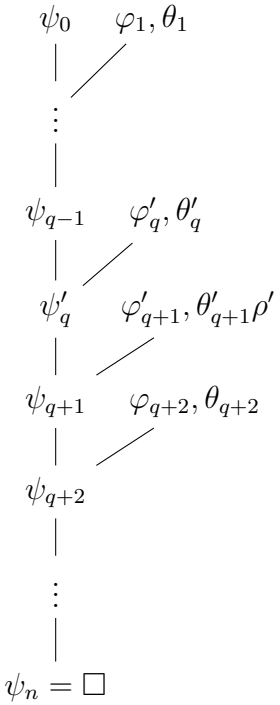
Demostración. Partimos de la SLD–refutación vía R



Vamos a ver que existe una SLD–derivación incompleta



de modo que existen sustituciones ρ' y ρ'' tales que $\theta'_q\theta'_{q+1} = \theta_q\theta_{q+1}\rho''$ y $\theta_q\theta_{q+1} = \theta'_q\theta'_{q+1}\rho'$. Tendremos entonces que



es una SLD–refutación con la misma respuesta σ.

Tenemos

- $\varphi^+_{q+1}\theta_{q+1} = \chi_j\theta_q\theta_{q+1}$, por hipótesis, y
- θ_q no afecta a las variables de φ^+_{q+1}, por la observación 2.16.

Por tanto, $\varphi^+_{q+1}\theta_q\theta_{q+1} = \chi_j\theta_q\theta_{q+1}$, i.e. φ^+_{q+1} y χ_j unifican. Por el teorema de unificación existe θ'_q u.m.g. de φ^+_{q+1} y χ_j, de modo que $\theta_q\theta_{q+1} = \theta'_q\rho$ para alguna sustitución ρ.

Por la observación 2.16, $\mathrm{Var}(\varphi_{q+1}) \cap \mathrm{Var}(\varphi_q) = \varnothing$. Entonces θ'_q no afecta a las variables de φ^+_q, por lo que

$$\varphi^+_q\rho = \varphi^+_q\theta'_q\rho = \varphi^+_q\theta_q\theta_{q+1} = \chi_i\theta_q\theta_{q+1} = \chi_i\theta'_q\rho,$$

i.e. φ^+_q y $\chi_i\theta'_q$ unifican. Por el teorema de unificación existe θ'_{q+1} u.m.g. de φ^+_q y $\chi_i\theta'_q$, de modo que $\rho = \theta'_{q+1}\rho'$ para alguna sustitución ρ'.

Ya hemos visto que $\theta_q\theta_{q+1} = \theta'_q\rho = \theta'_q\theta'_{q+1}\rho'$.

Como ya hemos dicho, $\mathrm{Var}(\varphi_{q+1}) \cap \mathrm{Var}(\varphi_q) = \varnothing$ por la observación 2.16, de modo que θ'_q no afecta a las variables de φ_q^+. Por ser además θ'_{q+1} u.m.g. de φ_q^+ y $\chi_i\theta'_q$ tenemos

$$\varphi_q^+\theta'_q\theta'_{q+1} = \varphi_q^+\theta'_{q+1} = \chi_i\theta'_q\theta'_{q+1}.$$

Ahora, como θ_q es u.m.g. de φ_q^+ y χ_i, existe γ sustitución tal que $\theta'_q\theta'_{q+1} = \theta_q\gamma$. Entonces

$$\chi_j\theta_q\gamma = \chi_j\theta'_q\theta'_{q+1} = \varphi_{q+1}^+\theta'_q\theta'_{q+1} = \varphi_{q+1}^+\theta_q\gamma = \varphi_{q+1}^+\gamma.$$

Por tanto por el teorema de unificación aplicado a γ tenemos $\gamma = \theta_{q+1}\rho''$ para alguna sustitución ρ'', de modo que $\theta'_q\theta'_{q+1} = \theta_q\gamma = \theta_q\theta_{q+1}\rho''$. □

Como corolario del lema de intercambio 2.33 tenemos el resultado clave de esta subsección.

Teorema 2.34 (de independencia de las reglas de computación)**.** *Sea P un programa lógico, ψ un objetivo y R una regla de computación. Supongamos que existe una SLD–refutación de $P \cup \{\neg\psi\}$ vía R con respuesta computada σ. Entonces para cualquier regla de computación R' existe una SLD–refutación de $P \cup \{\neg\psi\}$ vía R' con respuesta computada σ', de modo que $\psi\sigma$ es una variante de $\psi\sigma'$.*

Demostración. Si la longitud de la SLD–refutación vía R es n, aplicar $n-1$ veces el lema de intercambio 2.33. □

3. Implementación de Programas Lógicos: Prolog

Prolog es un lenguaje de programación basado en el mecanismo de ejecución de la Programación Lógica. En esta memoria trabajaremos con la implementación SWI–Prolog.

En esta sección introduciremos la sintaxis de Prolog que utiliza SWI–Prolog, explicaremos cómo implementa la SLD-resolución y presentaremos algunos ejemplos.

3.1. SWI–Prolog

3.1.1. Reglas sintácticas básicas

- **Carácteres admitidos:**
  - Alfanuméricos:
    - Letras: `a`, ..., `z`, `A`, ... `Z`.
    - Dígitos: `0`, ..., `9`.
  - Especiales: `+ - * / ( ) [ ] , . ^ = \ < > : % ? _` ...
- **Estructuras:** sucesiones finitas de carácteres admitidos de alguno de los siguientes tipos:

$$\textbf{Estructuras} \begin{cases} \text{Simples} \begin{cases} \textbf{Variables} \\ \text{Constantes} \begin{cases} \textbf{Números} \\ \textbf{Átomos} \end{cases} \end{cases} \\ \textbf{Compuestas} \end{cases}$$

- **Variables:** sucesiones finitas de carácteres alfanuméricos o `_` que comiencen por una letra mayúscula o por el carácter `_`.
  Ejemplos: `X`, `X1`, `X11`, `Juan`, `FechaDeNacimiento`, `_`, `_aBCde3_5`
  Llamaremos a `_` la variable anónima. Tiene un comportamiento especial: diferentes apariciones de la variable anónima representan variables diferentes.
- **Números:** sucesiones finitas de dígitos, opcionalmente con el signo `-` delante para representar enteros, y opcionalmente con un punto `.` en la sucesión (ni al principio ni al final) para representar números de punto flotante.
  Ejemplos: `345`, `-543`, `6239.57`, `-456.2`
  No serían números: `7.`, `.56`, `-.785`
- **Átomos:** los hay de tres tipos:
  1. Sucesiones finitas de carácteres alfanuméricos o `_` que comiencen por una letra minúscula.
     Ejemplos: `d`, `d2`, `d_5`, `fecha`, `fecha_de_compra`, `jesucristo_superstar`

2. Sucesiones finitas de carácteres admitidos, incluyendo el espacio en blanco, entre comillas simples `'`.
   Ejemplos: `'Jesucristo Superstar'`, `'+'`, `'a a a a'`
3. Sucesiones finitas de carácteres especiales. Muchos ya tienen una función predefinida, por ejemplo `+`, `-`, `*`, `/`, `:-`, `?-`. Sólo se utilizan para propósitos muy concretos.

En SWI–Prolog una sucesión de carácteres que representa un átomo, y esa misma sucesión entre comillas simples representan el mismo átomo. Por ejemplo: `fecha` y `'fecha'` representan el mismo átomo.

- **Estructuras compuestas:** formadas a partir de otras estructuras (simples o no), de la forma

  `pred(arg1,arg2,...,argN)`

  con `pred` un átomo que llamaremos *predicado* u *operador*, y con `arg1,...,argN` estructuras, que llamaremos *argumentos*, separadas por comas y entre paréntesis.
  Ejemplos: `hombre(H)`, `aniversario(Persona,fecha(1,enero,1995))`, `fecha(DIA,marzo,1784)`, etc.

- **Hechos:** un hecho es una estructura seguida de un punto.
  Ejemplos: `hombre(juan). igual(X,X). 24.`
  Representan hechos, i.e. cláusulas de programa de la forma $\varphi \leftarrow$.

- **Reglas:** las reglas son de la forma `est1 :- est2,...,estN.`, donde cada `esti` es una estructura. `est1` es la cabeza de la regla y `est2,...,estN` es el cuerpo. Notamos que las reglas también acaban en punto.
  Ejemplos: `abuelo(X, Y) :- padre(X, Z), padre(Z, Y).`
  Representan cláusulas de programa que no son hechos.

- **Cláusulas:** hechos o reglas.

- **Objetivos:** un objetivo es una sucesión de una o más estructuras separadas por coma, precedida por `?-` y acabada en punto.
  Ejemplos: `?- comida(macarrones).` `?- padre(X,yo), abuelo(X,tu).` `?- X = 3+4.`

- **Programa:** sucesión finita de cláusulas.

Observación 3.1. Prolog considera el primer átomo de un hecho como un símbolo de predicado, y el resto de átomos como símbolos de función. Lo mismo ocurre para cada una de las estructuras `esti` que forman las reglas de Prolog.

3.1.2. Funcionamiento. Backtracking.

Dado un programa P y un objetivo $\psi = \exists(\chi_1 \wedge \cdots \wedge \chi_n)$, el intérprete de SWI–Prolog ejecuta el siguiente algoritmo:

```
1   void prolog(prog P, obj ψ)
2   {
3     sld_resolucion(ε, P, ψ, variables_de(ψ));
4     puts("false.");
5   }

6   void sld_resolucion(sust θ, prog P, obj ψ, list Vs)
7   {
8     if (ψ == □)
9     {
10      θ = restringe_a_las_variables_de(θ, ψ);
11      write(θ);
12      char c = getchar();  /*tecla pulsada por el usuario*/
13      if (c != 'espacio' or ';')
14      {
15        puts(".");
16        exit();
17      }
18      else  puts(";");
19    }
20    else
21    {
22      formula_atomica χ = formula_atomica_mas_a_la_izquierda_de(ψ);
23      for(cada cláusula φ de P, por el orden de aparición en P)
24      {
25        clausula φ' = estandarizar_clausula(φ, Vs);
26        Vs = añadir_a( Vs, variables_de(φ') );
27        formula_atomica φ+ = cabeza_de(φ');
28        formula φ− = cuerpo_de(φ');
29        if (unifican(χ, φ+) == true)
30        {
31          sust θ' = umg(χ, φ+);
32          obj ψ' = sustituir_en_por(χ, ψ, φ−);
33          sust θ'' = composicion_sustituciones(θ, θ');
34          sld_resolucion( θ'', P, ψ', Vs);
35        }
36      }
37    }
38  }
```

de modo que si el algoritmo encuentra una respuesta computada θ la función `write`

la muestra por pantalla (línea 11), y en caso de no encontrar una se muestra el mensaje "`false.`" (línea 4). Notamos que en caso de que la respuesta computada hallada sea ε, la función `write` (línea 11) se limitará a mostrar el mensaje "`true.`" por pantalla.

En el proceso de búsqueda de una respuesta computada se produce un fenómeno conocido como *backtracking*: en una llamada de la función `sld_resolucion`, cada vez que la condición del `if` de la línea 29 no es cierta, esa llamada de la función `sld_resolucion` termina; es entonces cuando el propio algoritmo *vuelve a la anterior* (backtracks) llamada de `sld_resolucion`, desecha la cláusula φ de P que desencadenó la llamada a `sld_resolucion`, escoge la siguiente cláusula de P y vuelve a intentarlo con ella.

Exactamente el mismo fenómeno se produce cuando, habiéndose hallado una respuesta computada, el usuario pulsa la tecla 'espacio' o '`;`' (líneas 13 a 18). Termina entonces la función `sld_resolucion` y el algoritmo continúa buscando más respuestas computadas.

Para evitar el backtracking en este último caso basta con que una vez encontrada la primera solución el usuario pulse una tecla distinta de 'espacio o '`;`'. También se puede utilizar el predicado predefinido `once/1`.

3.1.3. Predicados predefinidos

SWI–Prolog incorpora una serie de predicados predefinidos. La notación `pred/n`, con `n` un número natural positivo, indica que el predicado `pred` es `n`–ario. A continuación listamos algunos de ellos:

- `Est1 = Est2`: Unifica la estructura `Est1` con la estructura `Est2`. Es cierto si la unificación tiene éxito. También se puede escribir `=(Est1, Est2)`.
- `read(Est)`: Lee por consola una estructura de Prolog y la unifica con `Est`.
- `write(Est)`: Escribe por consola la estructura `Est` usando paréntesis y operadores donde corresponda.
- `once(Obj)`: Evita el backtracking al encontrar la primera respuesta computada (si existe alguna).

3.1.4. Negación por fallo

SWI–Prolog implementa mediante los dos predicados equivalentes `not(Obj)` y `\+(Obj)` la denominada *negación por fallo*, que funciona de la siguiente manera:

- El intérprete dará `true` para la fórmula `not(Obj)` si todos los cómputos para `Obj` terminan y dan fallo.
- El intérprete dará `false` si encuentra un cómputo de éxito para `Obj`.

- Si el intérprete entra en un bucle infinito al intentar probar `Obj`, i.e. si se queda indefinidamente intentando construir una SLD–derivación infinita, el predicado no devolverá `true` independientemente de si existe o no alguna SLD–refutación.

Notamos que el predicado `not/1` es una forma parcial de la negación lógica $\neg$.

3.1.5. Listas

Las listas son las únicas estructuras de datos que se utilizan en Prolog[3]. Una lista en Prolog es la estructura `[]` o una estructura de la forma `[e1,...,en]`, donde `e1,...,en` son los elementos que contiene la lista separados por comas. Los elementos de las listas de Prolog pueden ser estructuras cualesquiera: átomos, variables, ..., incluso otras listas.

Las listas no tienen una longitud prefijada. Para trabajar con listas, Prolog dispone del operador binario `|`. Veamos algunos ejemplos para entender cómo se utilizan:

- `[X|Xs]`: denota la lista tal que su primer elemento es `X` y `Xs` es la lista restante.
- `[X1,...,Xn|Xs]`: denota la lista tal que sus primeros elementos son (en este orden) `X1`,...,`Xn` y `Xs` es la lista restante.
- `[]`: denota la lista vacía.
- `[X]`: denota la lista formada por un solo elemento, el elemento `X`. Equivale a `[X|[]]`.
- `[X1,...,Xn]`: denota la lista formada por los elementos `X1`,...,`Xn`. Equivale a `[X1,...,Xn|[]]`.

Algunos de los predicados predefinidos para trabajar con listas que SWI–Prolog incorpora son:

- `length(L, N)`: Es cierto si `N` representa el número de elementos de la lista `L`. Se puede usar en ambas direcciones: para hallar el número de elementos de una lista o para producir una lista (que contenga variables) de `N` elementos.
- `same_length(L1,L2)`: Es cierto si las listas `L1` y `L2` tienen el mismo número de elementos.
- `member(Elem,L)`: Es cierto si `Elem` es un elemento de la lista `L`.
- `numlist(N,M,L)`: Es cierto si `L` es la lista `[N,N+1,...,M]` donde `N` es un entero menor o igual al entero `M`.

[3]Las listas también son las únicas estructuras de datos que se utilizan en otros lenguajes de programación declarativos como Haskell y Lisp.

- `permutation(L1,L2)`: Es cierto cuando la lista `L1` es una permutación de la lista `L2`.
- `select(Elem,L1,L2)`: Es cierto cuando la lista `L1` sin una aparición del elemento `Elem` es la lista `L2`.
- `append(L1,L2,L3)`: Es cierto cuando la lista `L3` es la concatenación de las listas `L1` y `L2`.
- `append(LdeL,L)`: Concatena una lista de listas. Es cierto cuando `LdeL` es una lista de listas y `L` es la concatenación de esas listas.
- `transpose(L1,L2)`: Transpone una lista de listas de la misma longitud.
- `maplist(Obj, L)`: Es cierto si el objetivo `Obj` puede aplicarse con éxito a todos los elementos de la lista `L`.
  Ejemplo: `maplist(permutation(L),[L1,L2,...,LN])`. Es cierto si `permutation(L,Li)` es cierto para todo `i`=1,...,`N`.
- `string_chars(S,C)`: Se usa para convertir un string (cadena de carácteres entre comillas dobles `"`) `S` en una lista `C` que tiene por elementos los átomos que cada carácter de `S` representa.
  Ejemplo: `?- string_chars("a5.+",X).` tiene por única respuesta computada `X = [a, '5', '.', +]`.

3.1.6. Predicados aritméticos predefinidos

El siguiente programa implementa la suma sobre los naturales:

```
1 suma(0, X, X) :- natural(X).
2 suma(s(X), Y, s(Z)) :- suma(X, Y, Z).
3 natural(0).
4 natural(s(X)) :- natural(X).
```

No obstante no es práctico: además de lo engorroso que resulta tener que escribir 4 como `s(s(s(s(0))))`, el coste computacional de este predicado `suma/3` es proporcional al primer argumento, mientras que cualquier procesador realizará la operación en una unidad de tiempo de CPU (si los números no superan una cierta constante).

Para evitar este contratiempo SWI–Prolog incorpora algunos predicados para tratar con cálculos aritméticos de forma eficiente. Enumeramos a continuación los más usados:

- `X is ea`: unifica `X` con el valor de la expresión aritmética `ea`. En el momento en el que `is/2` sea llamado, `ea` no puede contener variables no instanciadas que impidan calcular su valor numérico. También se puede escribir `is(X, ea)`.
  Ejemplos:

```
?- X is 3+5.
X = 8.
```

```
?- 8 is 3+5.
true.
```

```
?- 3+5 is 3+5.
false.
```

```
?- Y is X+1, X = 5.
ERROR: Arguments are not
sufficiently instantiated
```

```
?- X = 5, Y is X+1.
X = 5, Y = 6.
```

- =:=, >, <, =<, >=: predicados binarios para comparar expresiones aritméticas. Ambos argumentos no deben contener variables no instanciadas a un número cuando el predicado sea llamado.

Podemos ahora reescribir el predicado `suma/3`:

```
1 suma(X, Y, Z) :- Z is X+Y.
```

```
?- suma(3, 2, Z).
Z = 5.
```

No obstante al utilizar el predicado aritmético `is/2` surge una desventaja: el programa no se comporta como una verdadera relación. Por ejemplo, no podemos usar el programa para restar (cosa que sí que podíamos hacer con el programa anterior):

```
?- suma(3, X, 5).
ERROR: Arguments are not sufficiently instantiated
```

Esto pasará no sólo con el predicado `is/2` sino con cualquier predicado que no sea bidireccional, i.e. que necesite que alguno de sus argumentos esté instanciado y por tanto no se comporte como una verdadera relación. Por ello el uso de estos predicados se traduce a menudo en una pérdida de generalidad de nuestros programas. Para evitarlo utilizaremos (siempre que trabajemos con enteros) la librería CLP(FD).

3.1.7. Librería CLP(FD)

La librería CLP(FD) (Constraint Logic Programming over Finite Domains) nos proporciona alternativas bidireccionales a los predicados aritméticos predefinidos. Estas alternativas sólo nos servirán cuando trabajemos con enteros[4]. Además algunos de sus predicados nos serán útiles para resolver problemas combinatorios. En SWI–Prolog no viene activada por defecto. La activaremos añadiendo la línea

[4]Existen además las librerías CLP(Q), CLP(R) y CLP(B) para trabajar con racionales, números de punto flotante y valores booleanos respectivamente.

```
:- use_module(library(clpfd)).
```

al principio del programa.

Los predicados que utilizaremos de esta librería son:

- **Restricciones de pertenencia:** sirven para especificar el dominio de una variable.
  - `Var in Dom`: Se puede leer como "la variable `Var` pertenece al dominio `Dom`", i.e. `Var` $\in$ `Dom`.
    El dominio `Dom` es de una de las siguientes formas:
    - Entero: conjunto unitario formado por dicho entero.
    - `N..M`: conjunto de los enteros `I` tales que `N` $\leq$ `I` $\leq$ `M`. `N` debe ser un entero o el átomo `inf`, que denota $-\infty$. `M` debe ser un entero o el átomo `sup`, que denota $+\infty$.
    - `Dom1 \/ Dom2`: la unión de los dominios `Dom1` y `Dom2`.

    Ejemplos: `X in 4`, que leeríamos como `X` $\in \{4\}$. `X in -10..10`, que leeríamos como `X` $\in [-10, 10] \cap \mathbb{Z}$. `X in 4..sup`, que leeríamos como `X` es un entero mayor o igual a 4.
  - `L ins Dom`: los elementos de la lista `L` pertenecen al dominio `Dom`.
- **Restricciones aritméticas y combinatorias:** sirven para reducir el dominio de una variable mediante una condición:
  - Condiciones aritméticas: usaremos los siguientes predicados:
    - `#=/2`: reemplazará a `is/2` y a `=:=/2` cuando trabajemos con enteros.
    - `#\=/2`: reemplazará a `=\=/2` cuando trabajemos con enteros.
    - `#\=/2`, `#>/2`, `#</2`, `#=</2`, `#>=/2`: reemplazarán a `=\=/2`, `>/2`, `</2`, `=</2`, `>=/2` respectivamente cuando trabajemos con enteros.
  - Condiciones combinatorias: sólo usaremos `all_distinct(L)`, que podemos leer como "los elementos de la lista `L` son distintos dos a dos".
- **Predicados de enumeración:** una vez reducido el dominio de una variable estos predicados asignan a la variable valores de dicho dominio.
  - `indomain(V)`: asignamos a `V` todos los posibles valores de su dominio reducido mediante backtracking. El dominio reducido de `V` debe ser finito.
    Ejemplo:

    ```
    ?- V in 0..sup, V #< 3, indomain(V).
    V = 0 ;
    V = 1 ;
    V = 2.
    ```
  - `label(L)`: aplica `indomain/1` a cada una de las variables de la lista `L`.
    Ejemplo:

```
?- [X,Y] ins 0..2, X #< Y, label([X,Y]).
X = 0,
Y = 1 ;
X = 0,
Y = 2 ;
X = 1,
Y = 2.
```

Podemos reescribir el programa anterior para sumar dos enteros usando `#=/2` en vez de `is/2`:

```
1 use_module(library(clpfd)).
2 suma(X, Y, Z) :- Z #= X+Y.
```

De esta manera ya podemos usar el predicado para restar:

```
?- suma(3, X, 5).
X = 2.
```

3.2. Recursión: Programas para cálculos aritméticos

A pesar de que Prolog no es el mejor software para realizar cálculos aritméticos, los ejemplos de esta subsección nos permitirán ilustrar como Prolog es ideal para programar funciones y relaciones definidas de forma recursiva.

3.2.1. Factorial

Para empezar programaremos el ejemplo por antonomasia de función aritmética definible por recursión: la función factorial. La función factorial se define como el producto

$$n! := \prod_{k=1}^{n} k = 1 \cdot 2 \cdots (n-1) \cdot n.$$

También es posible definirla mediante la relación de recurrencia

$$n! := \begin{cases} 1 & \text{si } n = 0, \\ (n-1)! \cdot n & \text{si } n > 0. \end{cases}$$

En Prolog la programaríamos de forma recursiva de la siguiente manera:

```
1 fact(0, 1).
2 fact(N, F) :-
3     N > 0,
```

```
4       N1 is N - 1,
5       fact(N1, F1),
6       F is N * F1.
```

Ejemplos:

```
?- fact(5,X).
X = 120 .

?- fact(45,X).
X = 119622220865480194561963161495657715064383733760000000000 .
```

No obstante el programa se comporta como una función y no como una relación, i.e. dado un $y \in \mathbb{N}$ no nos dice si existe algún $x \in \mathbb{N}$ tal que $x! = y$:

```
?- fact(X,24).
ERROR: Arguments are not sufficiently instantiated
```

¿Podríamos modificar nuestro programa para que no nos diera error? El programa lógico P formado por las cláusulas

$$\begin{aligned}&\mathrm{Fact}(0,1)\\&\mathrm{Fact}(x,y) \leftarrow (x>0) \wedge (x' = x-1) \wedge (y = x \cdot y') \wedge (\mathrm{Fact}(x',y'))\end{aligned}$$

tiene por modelo mínimo de Herbrand justo lo que buscamos,

$$M_P = \{\mathrm{Fact}(x,y) \colon x \in \mathbb{N},\ y = x!\}.$$

Sirviéndonos de la librería CLP(FD) lo implementamos de forma muy fácil:

```
1   :- use_module(library(clpfd)).
2   fact(0, 1).
3   fact(X, Y) :- X #> 0, X1 #= X-1, Y #= X*Y1, Y #\= 0, fact(X1, Y1).
```

Ya podemos utilizar el programa en ambas direcciones, e incluso sin explicitar ninguna variable.

```
?- fact(5, Y).
Y = 120 .
```

```
?- fact(X, 20).
false.
```

```
?- fact(X, 6).
X = 3 .
```

```
?- fact(X, 1).
```

```
X = 0 ;
X = 1 .
```

```
?- fact(X, Y).
X = 0, Y = 1 ;
```

```
X = Y, Y = 1 ;
X = Y, Y = 2 ;
X = 3, Y = 6 ;
...
```

3.2.2. Fracciones continuas

Una *fracción continua* es una expresión de la forma

$$x = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{\ddots}}}}$$

donde a_0 es un entero y todos los demás números a_i son enteros positivos.

La sucesión $(a_k)_k$ se llama *desarrollo en fracción continua* de x, y la denotaremos por

$$x = [a_0; a_1, a_2, \dots].$$

Podemos considerar la aplicación

$$\begin{aligned} \text{Frac\_cont}\colon \mathbb{Z} \times \mathbb{Z}_+^n &\to \mathbb{R} \\ [a_0; a_1, \dots, a_n] &\mapsto x. \end{aligned}$$

Programemos esta función en Prolog. Representaremos un desarrollo finito en fracción continua mediante una lista de Prolog. Dado un tal desarrollo de $x \in \mathbb{Q}$ el programa calculará la expresión decimal de x.

```
1 frac_cont([A], A).
2 frac_cont([A|As], X) :- frac_cont(As, Y), X is A+1/Y.
```

```
?- frac_cont([0,4],X).
X = 0.25 .

?- frac_cont([1,2,2,2,2,2,2,2,2,2,2,2,2,2,2],X).
X = 1.4142135624272734 .
```

3.2.3. Sumas geométricas

Implementemos en Prolog la siguiente función: dado un $n \in \mathbb{N}$,

$$F(n) = \sum_{\substack{0 \le k \le n \\ n \text{ es par}}} \frac{1}{2^k} + \sum_{\substack{0 \le k \le n \\ n \text{ es impar}}} \frac{1}{3^k}.$$

El programa lógico P formado por las cláusulas

$$F(0,1)$$
$$F(n,y) \leftarrow (n>0) \wedge \text{Impar}(n) \wedge (n' = n-1) \wedge (y' = y - (1/3)^n) \wedge F(n',y')$$
$$F(n,y) \leftarrow (n>0) \wedge \quad \text{Par}(n) \quad \wedge (n' = n-1) \wedge (y' = y - (1/2)^n) \wedge F(n',y')$$

donde $\text{Par}(n)$ e $\text{Impar}(n)$ son ciertos si n es un natural par e impar respectivamente, tiene por modelo mínimo de Herbrand

$$M_P = \{F(x,y) \colon x \in \mathbb{N},\ y = F(x)\}.$$

Lo implementamos en Prolog de la siguiente manera:

```
1  f(0,1).
2  f(N,Y) :- N>0, mod(N,2)=:=0, N1 is N-1, f(N1,Y1), Y is Y1+(1/2)^N.
3  f(N,Y) :- N>0, mod(N,2)=:=1, N1 is N-1, f(N1,Y1), Y is Y1+(1/3)^N.
```

```
?- f(1,X).
X = 1.3333333333333333 .

?- f(9,X).
X = 1.7070248993420718 .
```

Para disminuir el número de operaciones aritméticas podemos utilizar un acumulador, evitando así calcular la potencia `N`–ésima en cada recursión:

```
1  fe(N,Y) :- sum(N, Y, _, _).

2  sum(0,1,1/2,1/3).
3  sum(N,Y,AcPar,AcImpar) :-
4    N>0, mod(N,2) =:= 0, N1 is N-1, sum(N1,Y1,AcPar0,AcImpar0),
5    Y is Y1 + AcPar0,   AcPar is AcPar0/2, AcImpar is AcImpar0/3.
6  sum(N,Y,AcPar,AcImpar) :-
7    N>0, mod(N,2) =:= 1, N1 is N-1, sum(N1,Y1,AcPar0,AcImpar0),
8    Y is Y1 + AcImpar0, AcPar is AcPar0/2, AcImpar is AcImpar0/3.
```

3.3. Estrategia generar–comprobar: n–reinas

El problema de las n–reinas consiste en colocar n reinas en un tablero de ajedrez $n \times n$ sin que se amenacen entre ellas. Recordamos que en el juego del ajedrez una reina amenaza a aquellas piezas que se encuentran en su misma fila, columna o diagonales.

El siguiente programa en Prolog lo resuelve de forma muy ineficiente utilizando la estrategia *generar–comprobar*:

```
1  reinas(N, Rs) :-
2      numlist(1, N, Ns), permutation(Ns, Rs), no_se_atacan(Rs).

3  no_se_atacan([R|Rs]) :- no_se_atacan(Rs), not(ataca(R, Rs)).
4  no_se_atacan([]).

5  ataca(X, Xs) :- ataca(X, 1, Xs).

6  ataca(X, N, [Y|_]) :- X is Y+N; X is Y-N.
7  ataca(X, N, [_|Ys]) :- N1 is N+1, ataca(X, N1, Ys).
```

Una distribución concreta de las `N` reinas en el tablero `N`×`N` se representa como la permutación de una lista de Prolog que contiene los números del 1 al `N`, de manera que el primer elemento de la lista es la columna donde colocaríamos la reina en la primera fila, el segundo elemento es la columna donde colocaríamos la reina en la segunda fila, etc.

La relación `reinas(N, Rs)` es cierta si la lista `Rs` representa una distribución de las `N` reinas que resuelva el problema. El programa hace lo siguiente:

- El predicado `numlist(1, N, Ns)` genera una lista `Ns` con los números del `1` al `N`.
- El predicado `permutation(Ns, Rs)` *genera* una permutación `Rs` de la lista `Ns`.
- El predicado `no_se_atacan(Rs)` *comprueba* que las `N` reinas no se amenacen entre sí.

Por como hemos representado las posibles distribuciones, dos reinas no pueden estar en la misma fila o columna, por eso solamente necesitaremos comprobar que no se amenazan diagonalmente. El predicado `no_se_atacan/1` se define de forma recursiva: las reinas de `[R|Rs]` no se atacan si la primera de las reinas `R` de la lista no amenaza diagonalmente a las otras reinas de la lista `Rs` y a su vez las reinas de `Rs` no se atacan entre sí.

`ataca(R, Rs)` es cierto si la primera reina `R` amenaza a alguna de las otras reinas. Para comprobar si ello ocurre nos basamos en el hecho de que una reina amenaza a una segunda reina que está a `N` filas de distancia de la primera si y sólo si la columna de la segunda es `N` unidades mayor o menor a la columna de la primera. Las diagonales se comprueban iterativamente mediante el predicado `ataca/3` hasta llegar al final del tablero.

Si en vez de comprobar una permutación completa, i.e. colocando las `N` reinas en el tablero, comprobamos cada reina en el momento en el que la colocamos conseguimos, sin abandonar la estrategia *generar–comprobar*, mejorar la eficiencia. Es lo que hacemos en el siguiente programa:

```
reinas(N, Rs) :- numlist(1, N, Ns), reinas(N, Ns, [], Rs).

reinas(N, RsPorColocar, RsColocadas, Rs) :-
    select(R, RsPorColocar, RsPorColocar1),
    not(ataca(R, RsColocadas)),
    reinas(N, RsPorColocar1, [R|RsColocadas], Rs).
reinas(N, [], Rs, Rs):- imprimir_tablero(N, Rs).

ataca(X, Xs) :- ataca(X, 1, Xs).

ataca(X, N, [Y|_]) :- X is Y+N.
ataca(X, N, [Y|_]) :- X is Y-N.
ataca(X, N, [_|Ys]) :- N1 is N+1, ataca(X, N1, Ys).

imprimir_tablero(_, []).
imprimir_tablero(N, [R|Rs]) :-
    write('|'), imprimir_fila(1,R,N), imprimir_tablero(N, Rs).

imprimir_fila(I,_,N) :- I > N, write('|'), nl.
imprimir_fila(I,R,N) :-
    I =:= R, write('R '), J is I+1, imprimir_fila(J,R,N).
imprimir_fila(I,R,N) :-
    I =< N, I \= R, write('_ '), J is I+1, imprimir_fila(J,R,N).
```

Ahora es el predicado predefinido `select/3` el que *genera* y `ataca/2` el que *comprueba*.

El predicado `imprimir_tablero/2` permite imprimir por pantalla las diferentes distribuciones válidas que vamos encontrando.

Por ejemplo,

```
?- reinas(4, X).
|_ _ R _ |
|R _ _ _ |
|_ _ _ R |
|_ R _ _ |
X = [3, 1, 4, 2] ;
|_ R _ _ |
|_ _ _ R |
|R _ _ _ |
|_ _ R _ |
X = [2, 4, 1, 3] .

?- reinas(8, X).
|_ _ _ R _ _ _ _ |
```

```
|_ R _ _ _ _ _ _ |
|_ _ _ _ _ _ R _ |
|_ _ R _ _ _ _ _ |
|_ _ _ _ _ R _ _ |
|_ _ _ _ _ _ _ R |
|_ _ _ _ R _ _ _ |
|R _ _ _ _ _ _ _ |
X = [4, 2, 7, 3, 6, 8, 5, 1] ;
|_ _ _ _ R _ _ _ |
|_ R _ _ _ _ _ _ |
|_ _ _ R _ _ _ _ |
|_ _ _ _ _ _ R _ |
|_ _ R _ _ _ _ _ |
|_ _ _ _ _ _ _ R |
|_ _ _ _ _ R _ _ |
|R _ _ _ _ _ _ _ |
X = [5, 2, 4, 7, 3, 8, 6, 1] ;
...
```

3.4. Combinatoria: Sudoku

Un Sudoku es un pasatiempo matemático que consiste en rellenar una cuadrícula de 9×9 celdas (81 casillas) dividida en subcuadrículas de 3×3 (también llamadas "bloques") con las cifras del 1 al 9 partiendo de algunos números ya dispuestos en algunas celdas de la cuadrícula. La dificultad radica en que la cuadrícula se debe rellenar de tal forma que cada columna, fila y bloque contenga cada número del 1 al 9 exactamente una vez.

El siguiente programa en Prolog nos permitirá resolver Sudokus:

```
1  :- use_module(library(clpfd)).

2  sudoku(S) :-
3      length(S,9), maplist(same_length(S), S),
4      append(S, Vs), Vs ins 1..9,
5      maplist(all_distinct, S),
6      transpose(S, T), maplist(all_distinct, T),
7      bloques(S, B), maplist(all_distinct, B),
8      maplist(label, S).

9  bloques([F1,F2,F3,F4,F5,F6,F7,F8,F9], B) :-
10     append([F1,F4,F7],B1),
11     append([F2,F5,F8],B2),
12     append([F3,F6,F9],B3),
13     bloques(B1,B2,B3,B).
14 bloques([],[],[],[]).
```

```
15  bloques([E1,E2,E3|C1],[E4,E5,E6|C2],[E7,E8,E9|C3],
16    [[E1,E2,E3,E4,E5,E6,E7,E8,E9]|B]) :-
17      bloques(C1,C2,C3,B).
```

Una cuadrícula `N`×`N` se representa mediante una lista de Prolog `S`. Cada elemento de la lista `S` se corresponde con una fila de la cuadrícula y es a su vez una lista de Prolog que contiene las celdas de dicha fila.

Veamos qué hace el programa. Cinco de las seis líneas que forman el cuerpo de la cláusula que define el predicado `sudoku/1` se pueden leer de forma casi declarativa:

- **Línea 3:** `S` es una lista de 9 listas de 9 elementos.
  Interpretación: la cuadrícula representada por `S` es 9×9.
- **Línea 4:** Los elementos de las listas de `S` son enteros del 1 al 9.
  Interpretación: las celdas de la cuadrícula representada por `S` contienen enteros del 1 al 9.
- **Línea 5:** Los elementos de cada lista de `S` son distintos dos a dos.
  Interpretación: las filas de la cuadrícula representada por `S` no contienen números repetidos.
- **Línea 6:** Si `T` es la lista que se obtiene al transponer `S`, los elementos de cada lista de `T` son distintos dos a dos.
  Interpretación: las columnas de la cuadrícula representada por `S` no contienen números repetidos.
- **Línea 7:** Si `B` es la lista que contiene las listas que se obtienen al extraer recursivamente los 3 primeros elementos de las tres primeras listas de `S`, los elementos de cada lista de `B` son distintos dos a dos.
  Interpretación: los bloques de la cuadrícula representada por `S` no contienen números repetidos.

Como todos los predicados que hemos utilizado se comportan como verdaderas relaciones, el predicado `sudoku/1` también se comportará como una relación, por lo que podremos utilizar el programa, además de para solucionar, para comprobar y generar Sudokus:

```
?- X1=[_, _, _, _, _, _, _, _, 1],
|  X2=[_, _, _, _, _, _, _, 2, 3],
|  X3=[_, _, 4, _, _, 5, _, _, _],
|  X4=[_, _, _, 1, _, _, _, _, _],
|  X5=[_, _, _, _, 3, _, 6, _, _],
|  X6=[_, _, 7, _, _, _, 5, 8, _],
|  X7=[_, _, _, _, 6, 7, _, _, _],
|  X8=[_, 1, _, _, _, 4, _, _, _],
|  X9=[5, 2, _, _, _, _, _, _, _],
```

```
|  sudoku([X1,X2,X3,X4,X5,X6,X7,X8,X9]).
X1 = [6, 7, 2, 9, 8, 3, 4, 5, 1],
X2 = [9, 5, 1, 4, 7, 6, 8, 2, 3],
X3 = [3, 8, 4, 2, 1, 5, 9, 7, 6],
X4 = [4, 6, 8, 1, 5, 9, 2, 3, 7],
X5 = [2, 9, 5, 7, 3, 8, 6, 1, 4],
X6 = [1, 3, 7, 6, 4, 2, 5, 8, 9],
X7 = [8, 4, 3, 5, 6, 7, 1, 9, 2],
X8 = [7, 1, 9, 8, 2, 4, 3, 6, 5],
X9 = [5, 2, 6, 3, 9, 1, 7, 4, 8].

?- X1=[1, 5, 6, 8, 9, 4, 3, 2, 7],
|  X2=[9, 2, 8, 7, 3, 1, 4, 5, 6],
|  X3=[4, 7, 3, 2, 6, 5, 9, 1, 8],
|  X4=[3, 6, 2, 4, 1, 7, 8, 9, 5],
|  X5=[7, 8, 9, 3, 5, 2, 6, 4, 1],
|  X6=[5, 1, 4, 9, 8, 6, 2, 7, 3],
|  X7=[8, 3, 1, 5, 4, 9, 7, 6, 2],
|  X8=[6, 9, 7, 1, 2, 3, 5, 8, 4],
|  X9=[2, 4, 5, 6, 7, 8, 1, 3, 9],
|  sudoku([X1,X2,X3,X4,X5,X6,X7,X8,X9]).
true.

?- X1=[_, _, _, _, _, _, _, _, _],
|  X2=[_, _, _, _, _, _, _, _, _],
|  X3=[_, _, _, _, _, _, _, _, _],
|  X4=[_, _, _, _, _, _, _, _, _],
|  X5=[_, _, _, _, _, _, _, _, _],
|  X6=[_, _, _, _, _, _, _, _, _],
|  X7=[_, _, _, _, _, _, _, _, _],
|  X8=[_, _, _, _, _, _, _, _, _],
|  X9=[_, _, _, _, _, _, _, _, _],
|  sudoku([X1,X2,X3,X4,X5,X6,X7,X8,X9]).
X1 = [1, 2, 3, 4, 5, 6, 7, 8, 9],
X2 = [4, 5, 6, 7, 8, 9, 1, 2, 3],
X3 = [7, 8, 9, 1, 2, 3, 4, 5, 6],
X4 = [2, 1, 4, 3, 6, 5, 8, 9, 7],
X5 = [3, 6, 5, 8, 9, 7, 2, 1, 4],
X6 = [8, 9, 7, 2, 1, 4, 3, 6, 5],
X7 = [5, 3, 1, 6, 4, 2, 9, 7, 8],
X8 = [6, 4, 2, 9, 7, 8, 5, 3, 1],
X9 = [9, 7, 8, 5, 3, 1, 6, 4, 2] ;
X1 = [1, 2, 3, 4, 5, 6, 7, 8, 9],
X2 = [4, 5, 6, 7, 8, 9, 1, 2, 3],
X3 = [7, 8, 9, 1, 2, 3, 4, 5, 6],
X4 = [2, 1, 4, 3, 6, 5, 8, 9, 7],
```

```
X5 = [3, 6, 5, 8, 9, 7, 2, 1, 4],
X6 = [8, 9, 7, 2, 1, 4, 3, 6, 5],
X7 = [5, 3, 1, 6, 4, 2, 9, 7, 8],
X8 = [6, 4, 8, 9, 7, 1, 5, 3, 2],
X9 = [9, 7, 2, 5, 3, 8, 6, 4, 1] ;
...
```

En el anexo C se puede consultar una versión más general del programa que soluciona, comprueba y genera Sudokus 4×4, 9×9, 16×16, ...y, en general, $n^2 \times n^2$ para $n \geq 1$.

3.5. Implementación de autómatas: Reconocedor de expresiones booleanas

En esta subsección programaremos un *analizador booleano* en Prolog que reconozca expresiones booleanas válidas.

Ejemplo: `(3+5*7<2)` es una expresión booleana válida (los paréntesis cuadran, no hay signos mal puestos, etc.). En cambio `)7+*.(` no es una expresión booleana válida.

El analizador constará de dos partes:

- **Analizador léxico:** dada una expresión booleana de entrada generará una palabra de salida conteniendo las categorías sintácticas que aparecen en la expresión. Lo modelizaremos mediante una máquina traductora.
- **Analizador sintáctico:** averiguará si el output del analizador léxico se corresponde con el de una expresión booleana válida. Lo modelizaremos mediante un autómata con pila.

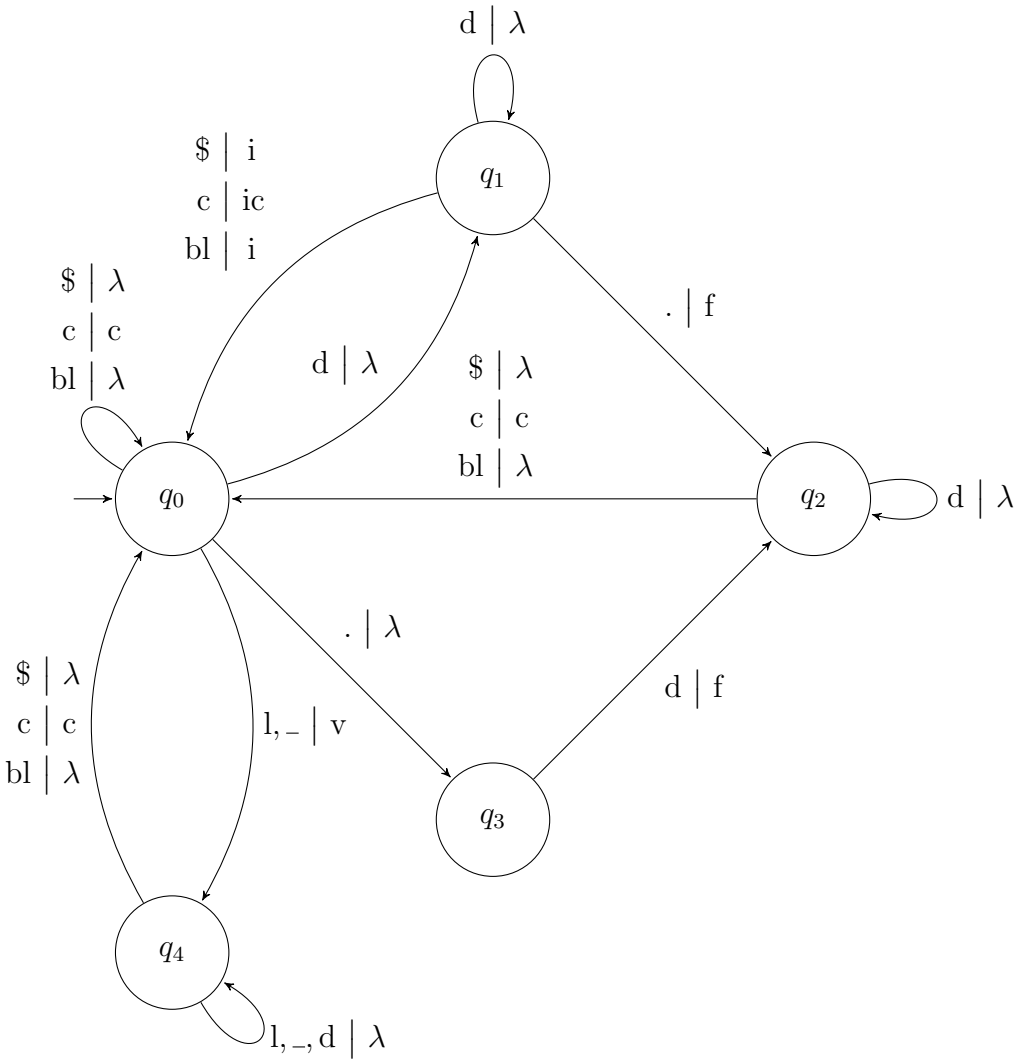
En el anexo B se pueden consultar las definiciones de *máquina traductora* y *autómata con pila* y los conceptos necesarios para comprenderlas.

Empecemos con el analizador léxico. Necesitaremos reconocer las siguientes categorías sintácticas:

- *integer*: enteros sin signo, i.e. cadenas de dígitos. La máquina traductora escribirá `i` en la cinta de salida.
- *float*: floats sin signo, i.e. cadenas de dígitos con un solo punto en cualquier lugar de la cadena. La máquina traductora escribirá `f` en la cinta de salida.
- *ID*: las variables aritméticas, que para nosotros serán cadenas formadas por letras, dígitos y '_' cuyo primer elemento no es un dígito. La máquina traductora escribirá `v` en la cinta de salida.
- Los siguientes símbolos gozarán de categoría propia: '+', '-', '*', '/', '(', ')', '=', '<' y '>'. La máquina traductora los escribirá tal cual en la cinta de salida.

Diseñemos la máquina traductora, que llamaremos M_1. Utilizaremos la siguiente notación para representar su grafo:

- _: el carácter de subrayado.
- bl: espacio en blanco.
- d: dígito.
- l: letra.
- $: denota el final de la expresión booleana de entrada ($\$ \in \Sigma$).
- c: cáracter del conjunto $\{+, -, *, /, (,), =, <, >\}$.
- λ: denota la palabra vacía.
- la coma no es en ningún caso un carácter válido.



Notamos que esta máquina traductora no es determinista en el sentido de que sus cómputos pueden bloquearse. No será un problema a la hora de implementarla con Prolog.

En general, para diseñar un programa en Prolog que simule el funcionamiento de una máquina traductora $M = (K, \Sigma, \Gamma, \delta, q_0)$ se procede como sigue:

1. Poner la cláusula

   ```
   trad(Xs, Ys) :- trad(q0, Xs, Ys).
   ```

   donde `q0` representa el estado inicial del autómata.

2. Para cada $(p, x) \in K \times (\Sigma \setminus \{\$\})$ del dominio de δ, si $\delta\big((p, x)\big) = (q, y_1 \dots y_n) \in K \times \Gamma^*$ poner la cláusula

   ```
   trad(p, [x|Xs], [y1,...,yn|Ys]) :- trad(q, Xs, Ys).
   ```

   donde `y1,...,yn` representa los n carácteres de la palabra $y_1 \dots y_n$.

3. Para cada $(p, \$) \in K \times \Sigma$ del dominio de δ, si $\delta\big((p, \$)\big) = (q, y_1 \dots y_n) \in K \times \Gamma^*$ poner la cláusula

   ```
   trad(p, [], [y1,...,yn]).
   ```

Siguiendo este procedimiento obtenemos un programa en Prolog que simula la máquina traductora M_1 anterior:

```
1  anal_lexico(S, Cs) :-
2      string_chars(S, Xs), trad(q0, Xs, Cs).

3  c(X) :- string_chars("()=<>+-*/", Ls), member(X, Ls).
4  d(X) :- string_chars("0123456789", Ls), member(X, Ls).
5  l(X) :- string_chars("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQ
6      RSTUVWXYZ_", Ls), member(X, Ls).
7  ld(X) :- string_chars("0123456789abcdefghijklmnopqrstuvwxyzABCDEF
8      GHIJKLMNOPQRSTUVWXYZ_", Ls), member(X, Ls).

9  trad(q0, [' '|Xs], Cs) :- trad(q0, Xs, Cs).
10 trad(q0, [C|Xs], [C|Cs]) :- c(C), trad(q0, Xs, Cs).
11 trad(q0, [D|Xs], Cs) :- d(D), trad(q1, Xs, Cs).
12 trad(q0, ['.'|Xs], Cs) :- trad(q3, Xs, Cs).
13 trad(q0, [L|Xs], [v|Cs]) :- l(L), trad(q4, Xs, Cs).
14 trad(q1, [D|Xs], Cs) :- d(D), trad(q1, Xs, Cs).
15 trad(q1, [' '|Xs], [i|Cs]) :- trad(q0, Xs, Cs).
16 trad(q1, [C|Xs], [i,C|Cs]) :- c(C), trad(q0, Xs, Cs).
17 trad(q1, ['.'|Xs], [f|Cs]) :- trad(q2, Xs, Cs).
18 trad(q2, [D|Xs], Cs) :- d(D), trad(q2, Xs, Cs).
19 trad(q2, [' '|Xs], Cs) :- trad(q0, Xs, Cs).
```

```
20  trad(q2, [C|Xs], [C|Cs]) :- c(C), trad(q0, Xs, Cs).
21  trad(q3, [D|Xs], [f|Cs]) :- d(D), trad(q2, Xs, Cs).
22  trad(q4, [L|Xs], Cs) :- ld(L), trad(q4, Xs, Cs).
23  trad(q4, [' '|Xs], Cs) :- trad(q0, Xs, Cs).
24  trad(q4, [C|Xs], [C|Cs]) :- c(C), trad(q0, Xs, Cs).

25  trad(q0, [], []).
26  trad(q1, [], [i]).
27  trad(q2, [], []).
28  trad(q4, [], []).
```

Ejemplo:

```
?- anal_lexico("(5*4.)-(-3)>.4", Cs).
Cs = ['(', i, *, f, ')', -, '(', -, i, ')', >, f].

?- anal_lexico("3+.", Cs).
false.
```

Sigamos con el analizador sintáctico. Lo modelizaremos mediante la siguiente gramática incontextual:

1. $S \longrightarrow \texttt{(}S\texttt{)} \mid E\texttt{=}E \mid E\texttt{<}E \mid E\texttt{>}E \mid E\texttt{<=}E \mid E\texttt{>=}E.$
2. $E \longrightarrow T\texttt{+}E \mid T\texttt{-}E \mid T.$
3. $T \longrightarrow F\texttt{*}T \mid F\texttt{/}T \mid F \mid \texttt{(+}F\texttt{)} \mid \texttt{(-}F\texttt{)}.$
4. $F \longrightarrow \texttt{(}E\texttt{)} \mid \texttt{i} \mid \texttt{f} \mid \texttt{v}.$

Aplicando el algoritmo B.21 del apéndice B obtenemos el autómata con pila equivalente

$$M_2 = (\{q_0, q_1\}, \Sigma, V \cup \Sigma, \Delta, q_0, \{q_1\}),$$

donde

$$\Sigma = \{\texttt{i}, \texttt{f}, \texttt{v}, \texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{=}, \texttt{<}, \texttt{>}, \texttt{(}, \texttt{)}\},$$
$$V = \{S, E, T, F\},$$

y Δ consta de

1. $\big((q_0, \lambda, \lambda), (q_1, S)\big)$
2. $\big((q_1, \lambda, S), (q_1, \texttt{(}S\texttt{)})\big)$
3. $\big((q_1, \lambda, S), (q_1, E\texttt{ = }E)\big)$
4. $\big((q_1, \lambda, S), (q_1, E\texttt{ < }E)\big)$
5. $\big((q_1, \lambda, S), (q_1, E\texttt{ > }E)\big)$
6. $\big((q_1, \lambda, S), (q_1, E\texttt{ <= }E)\big)$
7. $\big((q_1, \lambda, S), (q_1, E\texttt{ >= }E)\big)$
8. $\big((q_1, \lambda, E), (q_1, T\texttt{ + }E)\big)$

9. $\big((q_1, \lambda, E), (q_1, T \texttt{ - } E)\big)$
10. $\big((q_1, \lambda, E), (q_1, T)\big)$
11. $\big((q_1, \lambda, T), (q_1, F \texttt{ * } T)\big)$
12. $\big((q_1, \lambda, T), (q_1, F \texttt{ / } T)\big)$
13. $\big((q_1, \lambda, T), (q_1, F)\big)$
14. $\big((q_1, \lambda, T), (q_1, \texttt{(+}F\texttt{)})\big)$
15. $\big((q_1, \lambda, T), (q_1, \texttt{(-}F\texttt{)})\big)$
16. $\big((q_1, \lambda, F), (q_1, \texttt{(}E\texttt{)})\big)$
17. $\big((q_1, \lambda, F), (q_1, v)\big)$
18. $\big((q_1, \lambda, F), (q_1, i)\big)$
19. $\big((q_1, \lambda, F), (q_1, f)\big)$
20. $\big((q_1, \texttt{=}, \texttt{=}), (q_1, \lambda)\big)$
21. $\big((q_1, \texttt{<}, \texttt{<}), (q_1, \lambda)\big)$
22. $\big((q_1, \texttt{>}, \texttt{>}), (q_1, \lambda)\big)$
23. $\big((q_1, \texttt{+}, \texttt{+}), (q_1, \lambda)\big)$
24. $\big((q_1, \texttt{-}, \texttt{-}), (q_1, \lambda)\big)$
25. $\big((q_1, \texttt{*}, \texttt{*}), (q_1, \lambda)\big)$
26. $\big((q_1, \texttt{/}, \texttt{/}), (q_1, \lambda)\big)$
27. $\big((q_1, \texttt{v}, \texttt{v}), (q_1, \lambda)\big)$
28. $\big((q_1, \texttt{i}, \texttt{i}), (q_1, \lambda)\big)$
29. $\big((q_1, \texttt{f}, \texttt{f}), (q_1, \lambda)\big)$
30. $\big((q_1, \texttt{)}, \texttt{)}), (q_1, \lambda)\big)$
31. $\big((q_1, \texttt{(}, \texttt{(}), (q_1, \lambda)\big)$

En general, para diseñar un programa en Prolog que simule el funcionamiento de un autómata con pila $M = (K, \Sigma, \Gamma, \Delta, q_0, F)$ se procede como sigue:

1. Poner la cláusula

   ```
   aceptada(Xs) :- aceptada(q0, Xs, []).
   ```

   donde `q0` representa el estado inicial del autómata.

2. Para toda transición $((p, x_1 \dots x_k, a_1 \dots a_n), (q, b_1 \dots b_m)) \in \Delta$ poner la cláusula

   ```
   aceptada(p, [x1,...,xn|Xs], [a1,...,an|Ys]) :-
       aceptada(q, Xs, [b1,...,bm|Ys]).
   ```

3. Para todo $f \in F$ poner la cláusula

   ```
   aceptada(f, [], []).
   ```

Siguiendo este procedimiento obtenemos un programa en Prolog que simula el autómata con pila M_2 anterior:

```
1  anal_sintactico(Xs) :- aceptada(q0, Xs, []).

2  aceptada(q0, Xs, Ys) :- aceptada(q1, Xs, ['S'|Ys]).
3  aceptada(q1, Xs, ['S'|Ys]) :- aceptada(q1, Xs,  ['(','S',')'|Ys]).
4  aceptada(q1, Xs, ['S'|Ys]) :- aceptada(q1, Xs, ['E'|Ys]).
5  aceptada(q1, Xs, ['S'|Ys]) :- aceptada(q1, Xs, ['E',=,'E'|Ys]).
```

```
aceptada(q1, Xs, ['S'|Ys]) :- aceptada(q1, Xs, ['E',<,=,'E'|Ys]).
aceptada(q1, Xs, ['S'|Ys]) :- aceptada(q1, Xs, ['E',>,=,'E'|Ys]).
aceptada(q1, Xs, ['S'|Ys]) :- aceptada(q1, Xs, ['E',<,'E'|Ys]).
aceptada(q1, Xs, ['S'|Ys]) :- aceptada(q1, Xs, ['E',>,'E'|Ys]).
aceptada(q1, Xs, ['E'|Ys]) :- aceptada(q1, Xs, ['T',+,'E'|Ys]).
aceptada(q1, Xs, ['E'|Ys]) :- aceptada(q1, Xs, ['T',-,'E'|Ys]).
aceptada(q1, Xs, ['E'|Ys]) :- aceptada(q1, Xs, ['T'|Ys]).
aceptada(q1, Xs, ['T'|Ys]) :- aceptada(q1, Xs, ['F',*,'T'|Ys]).
aceptada(q1, Xs, ['T'|Ys]) :- aceptada(q1, Xs, ['F',/,'T'|Ys]).
aceptada(q1, Xs, ['T'|Ys]) :- aceptada(q1, Xs, ['F'|Ys]).
aceptada(q1, Xs, ['T'|Ys]) :- aceptada(q1, Xs, ['(',+,'F',')'|Ys]).
aceptada(q1, Xs, ['T'|Ys]) :- aceptada(q1, Xs, ['(',-,'F',')'|Ys]).
aceptada(q1, Xs, ['F'|Ys]) :- aceptada(q1, Xs, ['(','E',')'|Ys]).
aceptada(q1, Xs, ['F'|Ys]) :- aceptada(q1, Xs, [v|Ys]).
aceptada(q1, Xs, ['F'|Ys]) :- aceptada(q1, Xs, [i|Ys]).
aceptada(q1, Xs, ['F'|Ys]) :- aceptada(q1, Xs, [f|Ys]).
aceptada(q1, [=|Xs], [=|Ys]) :- aceptada(q1, Xs, Ys).
aceptada(q1,[<|Xs], [<|Ys]) :- aceptada(q1, Xs, Ys).
aceptada(q1,[>|Xs], [>|Ys]) :- aceptada(q1, Xs, Ys).
aceptada(q1,[+|Xs], [+|Ys]) :- aceptada(q1, Xs, Ys).
aceptada(q1,[-|Xs], [-|Ys]) :- aceptada(q1, Xs, Ys).
aceptada(q1,[*|Xs], [*|Ys]) :- aceptada(q1, Xs, Ys).
aceptada(q1,[/|Xs], [/|Ys]) :- aceptada(q1, Xs, Ys).
aceptada(q1,[v|Xs], [v|Ys]) :- aceptada(q1, Xs, Ys).
aceptada(q1,[i|Xs], [i|Ys]) :- aceptada(q1, Xs, Ys).
aceptada(q1,[f|Xs], [f|Ys]) :- aceptada(q1, Xs, Ys).
aceptada(q1,[')'|Xs], [')'|Ys]) :- aceptada(q1, Xs, Ys).
aceptada(q1,['('|Xs], ['('|Ys]) :- aceptada(q1, Xs, Ys).

aceptada(q1, [], []).
```

Ejemplos:

```
?- anal_sintactico(['(',i,-,'(',-,f,')',>,=,'(','(',f,
                    +,i,')',-,i,')',')']).
true .

?- anal_sintactico([f,*,/,i]).
false.
```

Finalmente, uniendo ambos códigos y añadiendo la cláusula

```
anal_booleano(S) :-
    once((anal_lexico(S, Xs), anal_sintactico(Xs))).
```

ya tenemos listo nuestro analizador booleano.

Ejemplo:

```
?- anal_booleano("(7-3)*8>=4/((-3)+12)").
true.

?- anal_booleano("4+++3=1").
false.
```

A. Teoría de retículos

A.1. Teorema de Tarski

Definición A.1 (Retículo completo)**.** Un conjunto parcialmente ordenado $(A, \leq)$ es un retículo completo cuando todo subconjunto de A tiene supremo e ínfimo.

Definición A.2 (Aplicación monótona creciente)**.** Sea $(A, \leq)$ un retículo completo y $T\colon A \to A$ una aplicación. Decimos que T es *monótona creciente* cuando para todo $a, b \in A$,

$$a \leq b \implies f(a) \leq f(b).$$

Definición A.3 (Mínimo punto fijo)**.** Sea $(A, \leq)$ un retículo completo y $T\colon A \to A$ una aplicación. Llamaremos el *mínimo punto fijo* de T a

$$\operatorname{mpf}(T) = \operatorname{mín}\{x \in A\colon x = T(x)\}.$$

Teorema A.4 (de Tarski)**.** *Si $(A, \leq)$ es un retículo completo y $T\colon A \to A$ es monótona creciente entonces T tiene un mínimo punto fijo* $\operatorname{mpf}(T)$ *tal que*

$$\operatorname{mpf}(T) = \operatorname{ínf}\{b \in A\colon T(b) \leq b\}.$$

Demostración. Sea $B = \{b \in A\colon T(b) \leq b\}$. Sea $a = \operatorname{ínf} B$, que existe por ser A retículo completo. Veamos que $a = T(a)$.
Para todo $b \in B$ tenemos

- $a \leq b$, por ser a ínfimo de B,
- $T(a) \leq T(b)$, por ser T monótona creciente,
- $T(b) \leq b$, por pertenecer b a B,

i.e. $T(a) \leq T(b) \leq b$, de modo que $T(a)$ es cota inferior de B. Como $a = \operatorname{ínf} B$, $T(a) \leq a$.
Ahora, como T es monótona creciente y $T(a) \leq a$ tenemos que $T(T(a)) \leq T(a)$, por tanto $T(a) \in B$. Entonces

- $T(a) \leq a$,
- $a \leq T(a)$, por ser $a = \operatorname{ínf} B$ y $T(a) \in B$,

i.e. $a = T(a)$.
Sea ahora $a' = \operatorname{ínf}\{b \in A\colon T(b) = b\}$, que existe por ser A retículo completo. Como a es un punto fijo, $a' \leq a$. Por otra parte, $\{b \in A\colon T(b) = b\} \subseteq B$ lo que implica que $a \leq a'$. Tenemos entonces que $a = \operatorname{mín}\{b \in A\colon T(b) = b\} = \operatorname{mpf}(T)$. □

A.2. Teorema de Kleene

Definición A.5. Sea $(A, \leq)$ un retículo completo y $T\colon A \to A$ monótona creciente. Para todo natural n definimos $T \uparrow n$ por:

- $T \uparrow 0 = \bot := \text{mín}\, A$,
- $T \uparrow (n+1) = T(T \uparrow n)$.

Ahora, definimos

$$T \uparrow \omega = \sup\{T \uparrow n : n \geq 0\}.$$

Definición A.6 (Conjunto dirigido)**.** Sea $(A, \leq)$ un retículo completo y $X \subseteq A$. Decimos que X es *dirigido* cuando todo subconjunto finito de X tiene una cota superior en X.

Definición A.7 (Aplicación continua)**.** Sea $(A, \leq)$ un retículo completo y $T\colon A \to A$ una aplicación. Decimos que T es *continua* cuando para todo subconjunto dirigido X de T,

$$T(\sup X) = \sup T[X].$$

Necesitamos dos lemas previos para enunciar y demostrar el teorema de Kleene.

Lema A.8. *Si $(A, \leq)$ es un retículo completo y $T\colon A \to A$ es continua entonces T es monótona creciente.*

Demostración. Sean $x, y \in A$. Tenemos que

$$x \text{ e } y \text{ son comparables} \iff \sup\{x, y\} \in \{x, y\}.$$

En particular,

$$x \leq y \iff \sup\{x, y\} = y.$$

Supongamos $x \leq y$. Como $\{x, y\}$ es dirigido y T es continua,

$$T(y) = T(\sup\{x, y\}) = \sup T\,[\{x, y\}] = \sup\{T(x), T(y)\}.$$

Por tanto,

$$T(x) \leq T(y).$$

□

Lema A.9. *Para todo x tal que $0 \leq x \leq \omega$,*

$$T \uparrow x \leq \text{mpf}(T).$$

Demostración. Por inducción:

- Si $x = 0$, $T \uparrow 0 = \text{ínf}\, A \leq \text{mpf}(T) \in A$.

- Si $0 < x < \omega$ y $\forall y < x$, $T \uparrow y \leq \mathrm{mpf}(T)$, como T es monótona creciente tenemos
$$T \uparrow x = T(T \uparrow (x-1)) \leq T(\mathrm{mpf}(T)) = \mathrm{mpf}(T).$$
- Si $x = \omega$ y $\forall y < \omega$, $T \uparrow y \leq \mathrm{mpf}(T)$, entonces
$$T \uparrow \omega = \sup\{T \uparrow n \colon n < \omega\} \leq \mathrm{mpf}(T).$$

□

Teorema A.10 (Kleene)**.** *Si $(A, \leq)$ es un retículo completo y $T \colon A \to A$ es continua entonces*
$$\mathrm{mpf}(T) = T \uparrow \omega.$$

Demostración. Sea $g = T \uparrow \omega = \sup\{T \uparrow n \colon n \geq 0\}$. Por el lema A.9 tenemos que $g \leq \mathrm{mpf}(T)$. Si vemos que g es un punto fijo de T habremos acabado. En efecto,
$$\begin{aligned} T(g) &= T(\sup\{T \uparrow n \colon n \geq 0\}) \\ &= \sup\{T(T \uparrow n) \colon n \geq 0\}, \text{ por ser } T \text{ continua} \\ &= \sup\{T \uparrow n \colon n \geq 0\} = g. \end{aligned}$$

□

B. Teoría de autómatas

B.1. Lenguajes

Definición B.1. Sea Σ un conjunto finito y no vacío de símbolos, que llamaremos *alfabeto*. Definimos una *palabra* sobre un alfabeto Σ como una secuencia finita de símbolos de Σ. Definimos

$$\Sigma^* = \{x : x \text{ es una palabra sobre el alfabeto } \Sigma\}.$$

Notación B.2. Denotaremos por λ a la palabra vacía (sin símbolos) sobre un alfabeto.
Si $x, y \in \Sigma^*$, denotaremos por xy a la yuxtaposición de x e y.

Definición B.3 (Lenguaje)**.** Un *lenguaje* L es un subconjunto de palabras sobre un alfabeto Σ, i.e. $L \subseteq \Sigma^*$.

Uno de los problemas que la teoría de autómatas trata de resolver es el de reconocer cuándo una palabra x pertenece a un lenguaje $L \subseteq \Sigma^*$.

B.2. Autómatas deterministas e indeterministas

Un autómata tiene asociada una cinta de entrada dividida en celdas. Suponemos que la cinta es infinita hacia la derecha. La cinta tiene asociada un puntero, que en un instante dado señala a una celda de la cinta. El puntero no puede moverse a la izquierda ni escribir en las celdas de la cinta. En cada paso de cómputo el puntero se mueve una celda a la derecha.

El autómata tiene asociada una "unidad de control", que en un instante dado se encuentra en un cierto estado.

Inicialmente, tenemos una palabra situada al comienzo de la cinta y el puntero señala a la primera celda.

Definición B.4 (Autómata determinista)**.** Un *autómata determinista* (AD) es una estructura $M = (K, \Sigma, \delta, q_0, F)$ donde:

1. K es un conjunto finito y no vacío de estados.
2. Σ es un alfabeto, el *alfabeto de entrada*.
3. $\delta\colon K \times \Sigma \to K$ es una aplicación que llamamamos *función de transición*.
4. $q_0 \in K$ es el *estado inicial*.
5. $F \subseteq K$ es el conjunto de *estados aceptadores* (o finales).

Llamamos *símbolo actual* al símbolo accesible en la cinta a través del puntero. En un *paso de cómputo* de M, se aplica la función δ al par (p, a) donde p es el estado actual y a es el símbolo actual. Lo denotaremos por

$$pax \vdash \delta(p, a)x \quad \text{con } x \in \Sigma^*.$$

Definición B.5 (Palabras reconocidas)**.** Sea $M = (K, \Sigma, \delta, q_0, F)$ un AD. Una palabra $x \in \Sigma^*$ es reconocida por M si existen pasos de cómputo tales que

$$q_0 x \vdash \cdots \vdash q$$

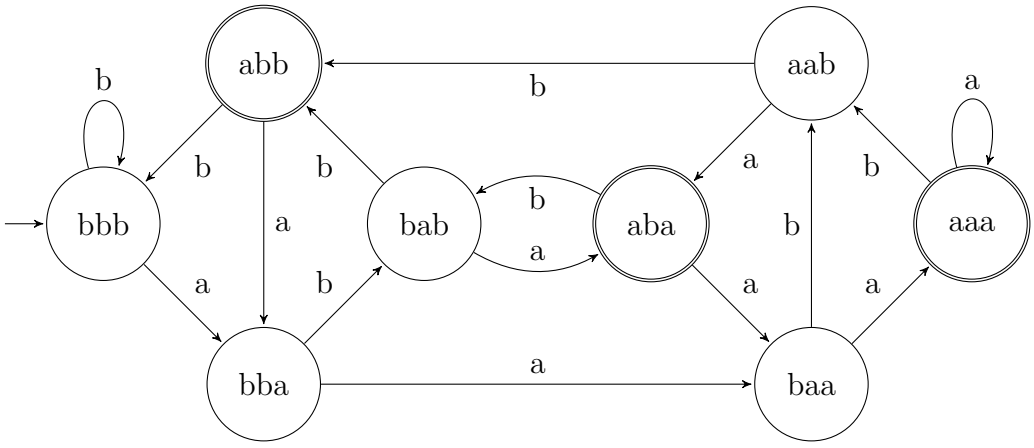
donde $q \in F$, i.e. q es un estado aceptador.
Definimos

$$L(M) = \{x \in \Sigma^* : x \text{ es reconocida por } M\}.$$

Representaremos a un AD mediante un grafo de la siguiente manera:

1. Los nodos del grafo son los estados del autómata.
2. Si $\delta(q, a) = p$, dibujamos un arco de q a p con etiqueta a.
3. Se marca con una flecha el estado inicial y con un doble círculo cada estado final.

Ejemplo B.6. Un AD que reconoce las palabras de $\{a, b\}^*$ que tienen una a en la antepenúltima posición. Los estados están etiquetados con las tres últimas letras leídas hasta el momento por el autómata, y en caso de que se hayan leído menos de tres letras se añaden b's a la izquierda.



Los autómatas indeterministas son equivalentes a los deterministas, pero permiten ser representados de forma más simple, a menudo requiriendo menos estados.

Definición B.7 (Autómata indeterminista)**.** Un *autómata indeterminista* (AI) es una estructura $M = (K, \Sigma, \Delta, q_0, F)$ donde K, Σ, q_0 y F son como en la definición de autómata determinista, y Δ es un subconjunto de $K \times (\Sigma \cup \{\lambda\}) \times K$.
Si p es el estado actual, $v \in \Sigma \cup \{\lambda\}$, $x \in \Sigma^*$ y $q \in K$ entonces

$$pvx \vdash qx$$

será un paso de cómputo de M si $(p, v, q) \in \Delta$.

Observación B.8. Al contrario que con los AD, dado un AI M, un estado actual p y una palabra x puede haber varios pasos de cómputo posibles, o no haber ninguno.

Definición B.9 (Palabras reconocidas)**.** Sea $M = (K, \Sigma, \Delta, q_0, F)$ un AI. Una palabra $x \in \Sigma^*$ es reconocida por M si existen pasos de cómputo tales que

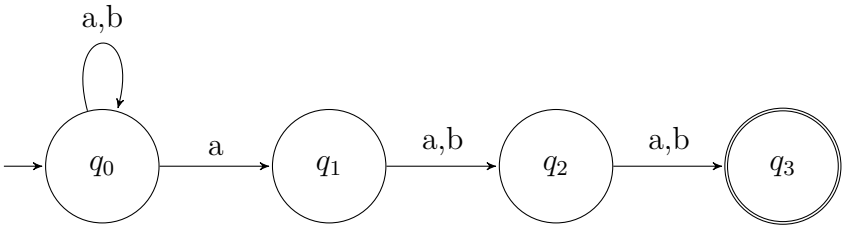
$$q_0 x \vdash \cdots \vdash q$$

donde $q \in F$, i.e. q es un estado aceptador.
Definimos

$$L(M) = \{x \in \Sigma^* \colon x \text{ es reconocida por } M\}.$$

Representaremos a un AI mediante un grafo de igual forma que haríamos con un autómata determinista.

Ejemplo B.10. Un AI que reconoce las mismas palabras que el AD del ejemplo B.6:



Sin embargo este AI tiene la mitad de estados que el anterior AD.

Teorema B.11. *Para todo lenguaje L,*

$$\textit{existe un AD } M \textit{ tal que } L = L(M) \iff \textit{existe un AI } M' \textit{ tal que } L = L(M').$$

Demostración. Ver [4, Teorema 2.3.1]. □

A pesar de que los AI's son más fácilmente representables, el indeterminismo puede darnos problemas a la hora de implementarlos. Se puede consultar en [4, Teorema 2.3.1] un algoritmo que nos permite eliminar el indeterminismo de un AI, i.e. obtener un AD equivalente a un AI dado.

B.3. Máquinas traductoras

Una máquina traductora es muy similar a un AD, salvo que su propósito no es aceptar palabras o lenguajes sino transformar la palabra de entrada en una palabra de salida. Informalmente, su unidad de control empieza en un estado inicial y se va moviendo de estado en estado según el símbolo de entrada de modo idéntico a un AD. La principal diferencia radica en que en cada paso de cómputo se escribe en una cinta de salida una palabra de cero o mayor longitud dependiendo del estado actual y del símbolo de entrada.

Definición B.12 (Máquina traductora)**.** Una *máquina traductora* es una estructura $M = (K, \Sigma, \Gamma, \delta, q_0)$ donde

1. K, Σ, q_0 son como en la definición de autómata determinista.

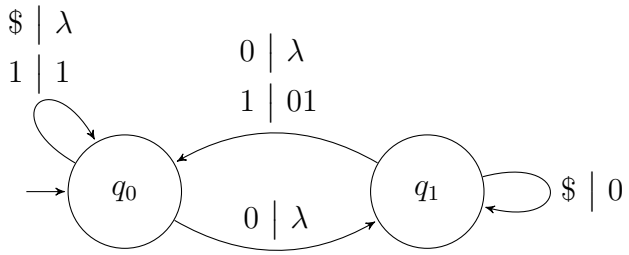
2. Γ es el alfabeto de salida.

3. $\delta\colon K \times \Sigma \to K \times \Gamma^*$ es la *función de transición*.

Observación B.13. Si necesitamos que la máquina escriba una palabra en la cinta de salida al acabar de leer la palabra de entrada añadiremos un símbolo auxiliar a Σ que originalmente no esté en Σ para representar el final de las palabras de la cinta de entrada.

Una máquina traductora se puede representar mediante un grafo igual al de un AD, salvo que la etiquetas de los arcos del grafo serán de la forma a $\mid$ w, i.e. si se lee la letra a en la cinta de entrada se escribe la palabra w en la cinta de salida.

Ejemplo B.14. Una máquina traductora que elimina las parejas de ceros de las palabras sobre el alfabeto $\{0, 1\}$:



El carácter \$ representa el final de las palabras de entrada.

B.4. Autómatas con pila y Gramáticas incontextuales

Se puede demostrar que los autómatas deterministas e indeterministas sólo reconocen un subconjunto de los lenguajes recursivos[5]. Los autómatas con pila los extienden en este sentido.

Definición B.15 (Autómata con pila)**.** Un *autómata con pila* (AP) es una estructura $M = (K, \Sigma, \Gamma, \Delta, q_0, F)$ donde:

1. K es un conjunto finito y no vacío de estados.

2. Σ es el *alfabeto de la cinta*.

3. Γ es el *alfabeto de la pila*.

4. $q_0 \in K$ es el *estado inicial*.

5. $F \subseteq K$ es el conjunto de *estados aceptadores*.

[5]Los lenguajes recursivos son aquellos $L \subseteq \Sigma^*$ tales que existe un algoritmo que dada una palabra $x \in \Sigma^*$ decide si $x \in L$ o $x \notin L$. Un lenguaje es recursivo si y sólo si existe una máquina de Turing que lo reconoce.

6. Δ es un subconjunto finito de

$$(K \times \Sigma^* \times \Gamma^*) \times (K \times \Gamma^*).$$

En un *paso de cómputo* se aplica una transición $((p, a, b), (q, x))$ de Δ y $a, b \neq \lambda$ de modo que

1. se pasa del estado p al estado q,
2. se lee la palabra a en la cinta de entrada, y
3. en la pila se reemplaza la palabra b por la palabra x de modo que x estará en la cima de la pila.

Lo denotaremos por

$$(by)p(az) \vdash (xy)pz, \quad y \in \Gamma^*,\ z \in \Sigma^*.$$

Observación B.16. Un AP puede ser indeterminista, i.e. dado un AP M, un estado actual p y una palabra x puede haber varios pasos de cómputo posibles, o no haber ninguno.

Definición B.17 (Palabras reconocidas)**.** Sea $M = (K, \Sigma, \Gamma, \Delta, q_0, F)$ un AP. Una palabra $x \in \Sigma^*$ es reconocida por M si existen pasos de cómputo tales que

$$\lambda q_0 x \vdash \cdots \vdash \lambda q$$

donde $q \in F$, i.e. q es un estado aceptador.

Definiremos a continuación el concepto de gramática incontextual y veremos su relación con los AP's.

Definición B.18 (Gramática incontextual)**.** Una *gramática incontextual* (Gi) es una estructura $G = (V, \Sigma, P, S)$ donde:

1. V es un alfabeto, a cuyos elementos se les llama *variables*.
2. Σ es un alfabeto disjunto de V, a cuyos elementos se les llama *terminales*.
3. P es un subconjunto finito de $V \times (V \cup \Sigma)^*$, a cuyos elementos se les llama *producciones* o *reglas*.
4. $S \in V$ es la variable inicial.

Si $(A, \alpha) \in P$ escribiremos $A \longrightarrow \alpha$, y si $(A, \alpha_1), \ldots, (A, \alpha_n) \in P$ escribiremos $A \longrightarrow \alpha_1 \mid \ldots \mid \alpha_n$.
Si $u, v \in (V \cup \Sigma)^*$ entonces

$$u \Rightarrow v$$

será una *derivación* de G si existen $x, y, w \in (V \cup \Sigma)^*$, $A \longrightarrow w \in P$ tales que $u = xAy$, $v = xwy$.

Definición B.19 (Palabras generadas)**.** Sea $G = (V, \Sigma, P, S)$ una Gi. Una palabra $x \in \Sigma^*$ es generada por G si existen derivaciones

$$S \Rightarrow \cdots \Rightarrow x.$$

Definimos

$$L(G) = \{x \in \Sigma^* \colon x \text{ es generada por } G\}.$$

Teorema B.20. *Para todo lenguaje L,*

$$\textit{existe un AP } M \textit{ tal que } L = L(M) \iff \textit{existe una Gi } G \textit{ tal que } L = L(G).$$

Demostración. Ver [4, Teorema 3.4.1]. □

A menudo será más fácil diseñar una Gi que nos genere un cierto lenguaje que diseñar el AP directamente. El siguiente algoritmo nos muestra como encontrar un AP equivalente a una Gi dada.

Algoritmo B.21 (para encontrar un AP equivalente a una Gi)**.** Sea $G = (V, \Sigma, P, S)$ una Gi. Definimos el AP $M = (\{q_0, q_1\}, \Sigma, V \cup \Sigma, \Delta, q_0, \{q_1\})$ donde Δ consta de las siguientes transiciones:

1. $((q_0, \lambda, \lambda), (q_1, S))$.
2. $((q_1, \lambda, A), (q_1, x))$ para cada $A \longrightarrow x$ en P.
3. $((q_1, a, a), (q_1, \lambda))$ para cada $a \in \Sigma$.

Teorema B.22. *Dada una Gi G, el algoritmo anterior define un AP M tal que*

$$L(G) = L(M).$$

Demostración. Ver [4, Lema 3.4.3]. □

C. Sudoku generalizado

Un Sudoku generalizado es un pasatiempo matemático que consiste en rellenar una cuadrícula de $n^2 \times n^2$ celdas (n^4 casillas) dividida en subcuadrículas de $n \times n$ (también llamadas "bloques") con los enteros del 1 al n^2 de modo que cada columna, fila y bloque contengan cada entero del 1 al n^2 exactamente una vez. Notamos que el caso $n = 3$ corresponde al Sudoku 9×9 estándar.

El siguiente programa en Prolog nos permitirá resolver Sudokus generalizados:

```
1  :- use_module(library(clpfd)).

2  sudoku(S) :-
3      N in 1..sup, N2 #= N*N,
4      length(S,N2), maplist(same_length(S), S),
5      append(S, Vs), Vs ins 1..N2,
6      cuadrado_latino(S),
7      bloques(N, S),
8      maplist(label, S).

9  cuadrado_latino(S) :-
10     maplist(all_distinct, S),
11     transpose(S, T), maplist(all_distinct, T).

12 bloques(_, []).
13 bloques(N, S) :-
14     length(S1, N), append(S1, S2, S),
15     f(N, S1),
16     bloques(N, S2).

17 f(_, S1) :- append(S1, []).
18 f(N, S1) :-
19     N2 #= N*N, length(B, N2), same_length(S1, S0),
20     sacar_bloque(N, S1, S0, B),
21     all_distinct(B),
22     f(N, S0).

23 sacar_bloque(_, [], [], []).
24 sacar_bloque(N, [F|S1], [X|S0], B) :-
25     length(P, N), append(P, X, F), append(P, Y, B),
26     sacar_bloque(N, S1, S0, Y).
```

Una cuadrícula `N2`×`N2` se representa mediante una lista de Prolog `S`. Cada elemento de la lista `S` se corresponde con una fila de la cuadrícula y es a su vez una lista de Prolog que contiene las celdas de dicha fila.

Expliquemos qué hace este programa:

- **Línea 1:** incluye la librería CLP(FD).

- **Línea 3:** calcula la dimensión `N2` de la cuadrícula calculando el cuadrado de `N`. Antes de esto hemos impuesto mediante el predicado `in/2` que `N` debe ser un entero entre `1` y `sup` $= +\infty$.

- **Línea 4:** imponemos mediante el predicado predefinido `length(S,N2)` que `S` debe contener `N2` elementos. Seguidamente el predicado `maplist/2` impone que el tamaño de cada elemento de `S` sea el mismo que el tamaño de la lista `S`, i.e. `N2`.
  Interpretación: la cuadrícula representada por `S` es `N2`×`N2`.
  Podría parecer más natural usar el predicado `length(lista, tamaño)` para comprobarlo, pero si queremos aprovechar `maplist/2` para aplicarla directamente sobre cada una de las filas de `S` necesitaríamos que `tamaño` fuese el primer argumento de `length/2` y `lista` el segundo. El predicado

  ```
  same_length(lista_de_tamaño_N2, lista)
  ```

  es equivalente a `length(lista, N2)` y nos evita el contratiempo.

- **Línea 5:** El predicado predefinido `append(S, Vs)` concatena las listas de `S` en una sola lista `Vs`, que pasará a contener todas las casillas de la cuadrícula representada por `S`. Seguidamente el predicado `Vs ins 1..N2` impone que los elementos de `Vs` pertenezcan al dominio de los enteros entre `1` y `N2`.
  Interpretación: las celdas de la cuadrícula representada por `S` contienen enteros entre 1 y `N2`.

- **Línea 6:** el predicado `cuadrado_latino/1` impone, con ayuda de los predicados `maplist/2`, `all_distinct/1` y `transpose/2`, que la cuadrícula representada por `S` sea un cuadrado latino.
  Interpretación: ni las filas ni las columnas de la cuadrícula representada por `S` contienen números repetidos.

- **Línea 7:** el predicado `bloques/2` impondrá que las casillas de los bloques de la cuadrícula no contengan números repetidos. Como en este programa `N2` no está prefijado, esta es la parte más complicada:

  - **Línea 14:** separa la lista `S` en dos sublistas `S1` y `S2`. `S1` contiene las `N` primeras filas de la cuadrícula y `S2` el resto.
  - **Línea 15:** `f(N, S1)` extrae los `N` bloques que forman estas `N` primeras filas de `S1` y comprueba que no contengan números repetidos.
    Para ello utiliza en la línea 20 el predicado `sacar_cuadro(N,S1,S0,B)`, que saca un cuadro de la lista `S1`, mete sus casillas en la lista `B`, y guarda en `S0` la lista `S1` sin el cuadro que ha quitado.
  - **Línea 16:** repite el proceso con `S2`.

- **Línea 8:** el predicado `label` de la librería CLP(FD) se utiliza para forzar al programa a unificar todas las variables de `S` con enteros entre `1` y `N2` antes de

mostrarnos una respuesta. De esta manera, si un Sudoku tuviese más de una solución posible el programa nos mostraría dichas soluciones de una en una mediante backtracking.

Ejemplos:

```
?- X1=[1,_,_,4],
|  X2=[_,4,1,_],
|  X3=[_,_,2,1],
|  X4=[2,_,4,_],
|  sudoku([X1,X2,X3,X4]).
X1 = [1, 2, 3, 4],
X2 = [3, 4, 1, 2],
X3 = [4, 3, 2, 1],
X4 = [2, 1, 4, 3] .

?- X1=[ _,11, 9, _, _,16,13, 4, _, _,14, _,10, 6,15, _],
|  X2=[ 4,12,15, _, 3, 6, _,11, _, 5, _, 1,16, 7,14, 2],
|  X3=[ 1, _, 6, _,15, 2, _, _,11, 9,10, _, _, _, 8, _],
|  X4=[ _,13, _, _, _, 1, _, _, 4, 6, _,15, _, _, _, _],
|  X5=[ _, _, _, _, _, _,15, _, 8, 1, 5, 3, _, 4,11, 7],
|  X6=[ 6, _, 1, _, _,12, 8, _, 9, _, _, 2, _, _, 3, _],
|  X7=[14, _, 4,13, 6, _, _, 3, _,12, 7,10, 8, _, 2, _],
|  X8=[ 3, 8, _, _, 4, 7, 2, _, 6, _, _, _, _,12,16, 5],
|  X9=[13, _, _,16, _, 8,14,10, 3, 4,15, _,12, 5, 1,11],
|  XA=[ _, _, _, 6, 2, _, _, 1,10, _,11, _,15, 3, _, 9],
|  XB=[ 7, _, _,12, _, 4, _,15, 5, _, 9,14, _, _, _, _],
|  XC=[10, _, _, 8, _, _,11, _, _, _, 1,12, 4, _,13,16],
|  XD=[ _, _, _, _, _, _, 7, _,15, 2, _, _, _, _,12, 3],
|  XE=[ _, _, 7, _, _,10, 6, _, 1, 8, _,13,11, _, 9,14],
|  XF=[ 8, 6, 5, _, _, 3, _, _,14, _, _, 9, _, _, _, _],
|  XG=[ _,16, _, 2, _, _, _,14, _,10, _, _, _, _, _, _],
|  sudoku([X1,X2,X3,X4,X5,X6,X7,X8,X9,XA,XB,XC,XD,XE,XF,XG]).
X1 = [2, 11, 9, 5, 8, 16, 13, 4, 12, 3, 14, 7, 10, 6, 15, 1],
X2 = [4, 12, 15, 10, 3, 6, 9, 11, 13, 5, 8, 1, 16, 7, 14, 2],
X3 = [1, 14, 6, 7, 15, 2, 5, 12, 11, 9, 10, 16, 3, 13, 8, 4],
X4 = [16, 13, 8, 3, 14, 1, 10, 7, 4, 6, 2, 15, 9, 11, 5, 12],
X5 = [12, 2, 16, 9, 10, 14, 15, 13, 8, 1, 5, 3, 6, 4, 11, 7],
X6 = [6, 7, 1, 11, 5, 12, 8, 16, 9, 15, 4, 2, 14, 10, 3, 13],
X7 = [14, 5, 4, 13, 6, 11, 1, 3, 16, 12, 7, 10, 8, 9, 2, 15],
X8 = [3, 8, 10, 15, 4, 7, 2, 9, 6, 14, 13, 11, 1, 12, 16, 5],
X9 = [13, 9, 2, 16, 7, 8, 14, 10, 3, 4, 15, 6, 12, 5, 1, 11],
XA = [5, 4, 14, 6, 2, 13, 12, 1, 10, 16, 11, 8, 15, 3, 7, 9],
XB = [7, 1, 11, 12, 16, 4, 3, 15, 5, 13, 9, 14, 2, 8, 10, 6],
XC = [10, 15, 3, 8, 9, 5, 11, 6, 2, 7, 1, 12, 4, 14, 13, 16],
XD = [11, 10, 13, 14, 1, 9, 7, 8, 15, 2, 6, 4, 5, 16, 12, 3],
```

```
XE = [15, 3, 7, 4, 12, 10, 6, 5, 1, 8, 16, 13, 11, 2, 9, 14],
XF = [8, 6, 5, 1, 13, 3, 16, 2, 14, 11, 12, 9, 7, 15, 4, 10],
XG = [9, 16, 12, 2, 11, 15, 4, 14, 7, 10, 3, 5, 13, 1, 6, 8] .

?- sudoku(S).
S = [[1]] ;
S = [[1, 2, 3, 4], [3, 4, 1, 2], [2, 1, 4, 3], [4, 3, 2, 1]] ;
S = [[1, 2, 3, 4], [3, 4, 1, 2], [2, 3, 4, 1], [4, 1, 2, 3]] ;
S = [[1, 2, 3, 4], [3, 4, 1, 2], [4, 1, 2, 3], [2, 3, 4, 1]] ;
S = [[1, 2, 3, 4], [3, 4, 1, 2], [4, 3, 2, 1], [2, 1, 4, 3]] ;
S = [[1, 2, 3, 4], [3, 4, 2, 1], [2, 1, 4, 3], [4, 3, 1, 2]] ;
S = [[1, 2, 3, 4], [3, 4, 2, 1], [4, 3, 1, 2], [2, 1, 4, 3]] ;
...
```

Referencias

[1] J. W. Lloyd: *Foundations of Logic Programming*, 2nd extended ed., Springer, 1987.

[2] L. Sterling; E. Shapiro: *The Art of Prolog*, 2nd Edition, MIT Press, 1994.

[3] I. Bratko: *Prolog: Programming for Artificial Intelligence*, 2nd ed., Addison-Wesley, 1994.

[4] Harry R. Lewis; Christos H. Papadimitriou: *Elements of the Theory of Computation*, 1st ed., 1981.

[5] Chin-Liang Chang; Richard C. Lee: *Symbolic Logic and Mechanical Theorem Proving*, Academic Press Inc, 1973.

[6] Uwe Schöning; *Logic for Computer Scientists*, Birkhäuser, 1989.

[7] H.-D. Ebbinghaus; J. Flum; W. Thomas: *Mathematical Logic*, 2nd ed., Springer, 1994.